Marco Fidel Vasquez - A00362057
Diego Alejandro - A00362738

REPORT OF ENGINEERING METHOD

PROBLEM IDENTIFICATION

Problem Necessities

To get the shortest path from the start to the exit of a labyrinth using two main criteria:

- To get the path that uses the minimum quantity of cells from the start to the exit.
- To get the path with the minimum difficulty from the start to the exit (each cell has its own difficulty and the difficulty of the path is the summation of all the cells' difficulty).

Problem Definition

- A person is trapped in a maze and needs our help to get out and get back with his family. The maze is built like a matrix but is not easy to escape because there's a lot of cells with traps that makes it difficult to go through them. The person needs to know what is the safest and the shortest path to get out of the maze .

Functional Requirements:

FR1: Create a matrix with 20 rows and 20 columns.

FR2: Recreate the maze in the matrix selecting the cells that are part of the paths of the maze

FR3: Find the shortest path to the exit.

FR4: Find the safest path to the exit

FR5: Mark with orange the cells that are part of the safest way out of the maze.

FR6: Mark with blue color the cells that are part of the shortest way out of the maze.

FR7: Generate automatically the difficulty of every cell.

Non Functional Requirements:

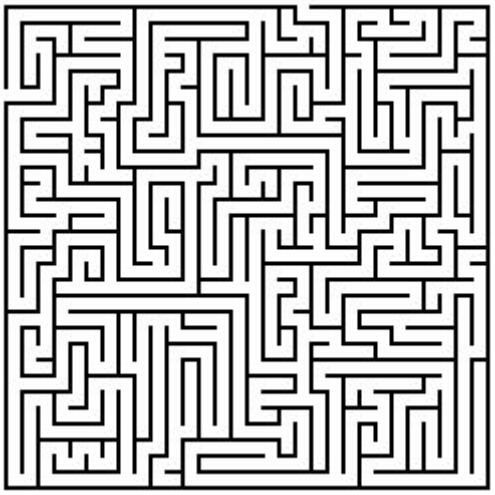
NFR1: Use a graph to represent the maze .

COMPILATION OF INFORMATION

Definitions

Maze[1]:

A maze is a path or collection of paths, typically from an entrance to a goal. The word is used to refer both to branching tour puzzles through which the solver must find a route, and to simpler non-branching ("unicursal") patterns that lead unambiguously through a convoluted layout to a goal. The pathways and walls in a maze are typically fixed, but puzzles in which the walls and paths can change during the game are also categorised as mazes or tour puzzles.



Matrix[2]: A matrix is a grid used to store or display data in a structured format. It is often used synonymously with a table, which contains horizontal rows and vertical columns.

POSSIBLE SOLUTIONS

Idea generation methodology used: Brainstorming.

Possible solutions for find the shortest path:

Solution 1: Use a Breadth-First Search algorithm to find the shortest path.

Solution 2: Use a Depth-First Search algorithm to find the shortest path.

Solution 3: Use a Dijkstra algorithm with ponderation 1 to find the shortest path.

Possible solutions for find the safest path:

Solution 1: Use a Dijkstra algorithm to find the safest path

---

[1] https://en.wikipedia.org/wiki/Maze

[2] https://techterms.com/definition/matrix

Solution 2: Use a Prim algorithm to find the safest path.

Solution 3: Use a Kruskal algorithm to find the safest path.

Solution 4: Use floyd-Warshall algorithm to find the safest path.

TRANSFORMING IDEA FORMULATION TO PRELIMINARY DESIGNS

Possible solutions for find the shortest path:

Solution 1: The Breadth-First Search is an algorithm that transverse a graph from a selected node to find all the vertices in the graph. This algorithm works visiting the vertices for layers, thus must visit the neighbour nodes of the selected node. Then move forward to the next level neighbour nodes.

Possible solutions for find the safest path:

Solution 1: The Dijkstra algorithm, given a graph and a source vertex in the graph, finds shortest paths from source to all vertices in the given graph. With an adjust this algorithm is useful to build the minimum cost way out of the maze.

Solution 3: Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if it has the least cost among all available options and does not violate MST properties. It's useful in this case because we can build a path with minimum cost.

DISCARDED IDEAS

Possible solutions for find the shortest path:

Solution 2: The Depth-First Search algorithm is useful when the problem that we are gonna solve has a unique solution, the main problem with this algorithm is that our problem has multiple solutions (multiple paths) so in order to solve it with this algorithm we should run it until it has travel through all the vertices and that solution would be very slow and very poorly optimized.

Solution 3: The Dijkstra algorithms would be a good solution for this problem, but that leads to use ponderation 1 in order to make that it works, each cell has its own difficulty so it would interfere with the use of dijkstra in this case. Besides, BST is way a better solution if we take in account the time complexity.

Possible solutions for find the safest path:

Solution 2: PRIM algorithm can not be used in this problem because a prerequisite to use prim is that the graph must be conex and in this problem the graph isn't conex.

Solution 4: Floyd-Warshall is not a good solution to this problem since we are not searching the safest path between every pair of cells but only the start and the exit. We could modify it to get just the path we want but it would be very slow and poorly optimized.

EVALUATION AND SELECTION OF THE BEST SOLUTION

solution	knowledge about the topic	Value for the client	Easy of development	Flexibility	Total	Approved/ Not approved
Possible solutions for find the shortest path						
solution 1	4	5	4	4	17	Approved
Possible solutions for find the safest path						
solution 1	3	5	3	4	15	Approved
solution 3	4	5	3	5	17	Approved

Knowledge about the topic: Our theoretical knowledge about the topic required to implement this solution in the program.

Value for the client: The value that the implementation of this solution that will visually and performance-wise give to the client, such as: Faster recognition, ease of use and resource allocation.

Ease of development: How easy and/or fast it is to develop the solution in code.

Flexibility: How flexible the code is in regards to future-proofing and fixing any errors that may arise in its use.

Solution chosen for find the shortest path:

We have chosen solution 1 because it is the only one which left after making a value judgment to know which solution would dock better with this problem.

Solution chosen for find the safest path:

We have chosen both options because we want to study and have a better understanding of these algorithms, that is why we have chosen to use both of these algorithms for our program.

TAD FOR GRAPH

TAD GRAPH		
A n-tuple <T,<T,integer>,<T,integer>….>, where the first one is the vertex and the other ones arte its edges and the weight of each edge Type or T is the name of the vertex, which could be an Integer,String, Double, etc. We call it T to generalize this parameter.		
true		
Primitive operations:		
CreateGraph:		Graph
addVertex:	Type	Graph
addEdge:	TypexType	Graph
addEdge:	TypexTypexweight	Graph
deleteVertex	Type	Graph
dfs:	Type	HashMap
bfs:	Type	HashMap
dijkstra:	Type	HashMap
floydWarshall:	True	matrix
prim:	True	Graph
kruskal:	True	Graph

CreateGraph()

Constructor operation

“creates a graph”

Pre: true

Pos: a new graph

AddVertex(Type)

Modifier operation

“adds avertex to the graph”

Pre: a graph must already exist

Pos: a new graph

AddEdge(TypexType)

modifier operation

“adds an edge to the graph”

Pre: a graph must already exist

Pos: a new graph

AddEdge(TypexTypexweight)

modifier operation

“adds an edge with weight to the graph”

Pre: a graph must already exist

Pos: a new graph

DFS(Type)

analyzer operation

“returns a minimum cost tree turned into a hashtable”

Pre: a graph must already exist

T_c = Type current, this is the current vertex.
T_p =Type previous, this is the vertex from which we can reach the current vertex.

Pos: {< T_c , T_p >,< T_c , T_p >,...}

BFS(Type)

analyzer operation

“returns a minimum cost tree turned into a hashtable”

Pre: a graph must already exist

T_c = Type current, this is the current vertex.
T_p =Type previous, this is the vertex from which we can reach the current vertex.

Pos: {< T_c , T_p >,< T_c , T_p >,...}

Dijkstra(Type)

analyzer operation

“return the minimum cost path from a given vertex to all the other vertices of the graph”

Pre: a graph must already exist

T_c = Type current, this is the current vertex.
T_p =Type previous, this is the vertex from which we can reach the current vertex.

Pos: {< T_c , T_p >,< T_c , T_p >,...}

floydWarshall()

analyzer operation

“return the minimum cost path between all the existing pairs of the graph”

Pre: a graph must already exist

Pos: matrix of integer

prim()

analyzer operation

“return the minimum spanning tree for connected graphs”

Pre: a graph must already exist and it must be connected

Pos: graph

kruskal()

analyzer operation

“return the minimum spanning tree for graphs”

Pre: a graph must already exist

Pos: graph

Test Design

Scenarios Setup

Name	Class	Scenario
setup1	Graph	An empty graph.
setup2	Graph	A connected graph with 5 vertices {San Francisco,Denver,Chicago,Atlanta,New York} .There is a visual of the graph in the annexed 1.1
setup3	Graph	A graph with 4 vertices {1,2,3,4}. There is a visual of the graph in the annexed 1.2
setup4	Graph	A graph with 6 vertices {a,b,c,d,e,}. There is a visual of the graph in the annexed 1.3
setup5	Graph	A graph with 8 vertices {r,s,t,u,v,w,x,y} With a few edges in it. There is a visual of the graph in the annexed 1.4
setup6	Graph	A graph with 5 vertices added {1,2,3,4,5}.
setup7	Graph	A connected graph with 5 vertices added {1,2,3,4,5}.
setup8	Graph	A graph with 6 vertices{u,v,w,x,y,z}. With a few edges in it. There is a visual of the graph in the annexed 1.5

Test objective:
To check if the addVertex method works properly by adding 5 vertices and then manually retrieve them and verify if they are added correctly.

Class	Method	Scenario	Input values	Result
Graph	addVertex	setup1	1	We look for the vertex on our graph and find the vertex 1 added.
Graph	addVertex	setup1	2	We look for the vertex on our graph and find the vertex 2 added.

Graph	addVertex	setup1	3	We look for the vertex on our graph and find the vertex 3 added.
Graph	addVertex	setup1	4	We look for the vertex on our graph and find the vertex 4 added.
Graph	addVertex	setup1	5	We look for the vertex on our graph and find the vertex 5 added.

Test objective:

Check if the addEdge method works properly by connecting the vertex 1 with all other vertices, then retrieve them and verify if they are correctly connected.

Class	**Method**	**Scenario**	**Input values**	**Result**
Graph	addEdge	setup6	1,2	The connection between vertex 1 and 2 was made successfully.
Graph	addEdge	setup6	1,3	The connection between vertex 1 and 3 was made successfully.
Graph	addEdge	setup6	1,4	The connection between vertex 1 and 4 was made successfully.
Graph	addEdge	setup6	1,5	The connection between vertex 1 and 5 was made successfully.

Test objective:

Check if the addEdge method with weight works properly by connecting the vertex 1 with all other vertices, then retrieve them and verify if they are correctly connected and its weights are correct.

Class	Method	Scenario	Input values	Result
Graph	addEdge	setup6	1,2,20	The connection between vertex 1 and 2 was made successfully and its weight is 20.
Graph	addEdge	setup6	1,3,1	The connection between vertex 1 and 3 was made successfully and its weight is 1.
Graph	addEdge	setup6	1,4,23	The connection between vertex 1 and 4 was made successfully and its weight is 23.
Graph	addEdge	setup6	1,5,54	The connection between vertex 1 and 5 was made successfully and its weight is 54.

Test objective:

Check if the deleteVertex method works properly by deleting one by one the vertices of the graph and verify if the connections were also correctly deleted.

Class	Method	Scenario	Input values	Result
Graph	deleteVertex	setup7	10	The vertex doesn't exist in the graph.

Graph	deleteVertex	setup7	5	The vertex 5 was correctly deleted with all the connections.
Graph	deleteVertex	setup7	4	The vertex 4 was correctly deleted with all the connections.
Graph	deleteVertex	setup7	3	The vertex 3 was correctly deleted with all the connections.
Graph	deleteVertex	setup7	2	The vertex 2 was correctly deleted with all the connections.
Graph	deleteVertex	setup7	1	The vertex 1 was correctly deleted with all the connections.
Graph	deleteVertex	setup7	1	The graph is empty.

Problem solution Test cases.

Test objective:

Check if the prim method returns the correct minimum spanning tree represented as a graph.

Class	Method	Scenario	Input values	Result
Graph	prim	setup2	none	A minimum spanning tree of the graph. We verify the vertices and the weights of the new graph.

				There is a visual of the result of the method in the annexed 2.1

Test objective:

Check if the kruskal method returns the correct minimum spanning tree represented as a graph.

Class	Method	Scenario	Input values	Result
Graph	kruskal	setup2	none	A minimum spanning tree of the graph. We verify the vertices and the weights of the new graph. There is a visual of the result of the method in the annexed 2.1

Test objective:

Check if the floydWarshall method returns a correct matrix representation of the graph with the minimum distance between every pair of vertices of the graph.

Class	Method	Scenario	Input values	Result
Graph	floydWarshall	setup3	none	A matrix with the minimum distance of every pair of vertices of the graph. We go through the matrix verifying the distances. There is a visual of the result of the method in the annexed 2.2

Test objective:

Check if the dijkstra method returns a correct hashtable with all the previous vertices of the current existing vertices of the graph.

Class	Method	Scenario	Input values	Result
Graph	dijkstra	setup4	a	A hashtable with the previous vertices of the existing vertices. We verify the previous of each vertex in the hashtable. There is a visual of the result of the method in the annexed 2.3

Test objective:

Check if the bfs method returns a minimum cost tree turned into a hashtable with all the previous vertices of the current existing vertices of the graph.

Class	Method	Scenario	Input values	Result
Graph	bfs	setup5	s	A hashtable with the previous vertices of the existing vertices. We verify the previous of each vertex in the hashtable. There is a visual of the result of the method in the annexed 2.4

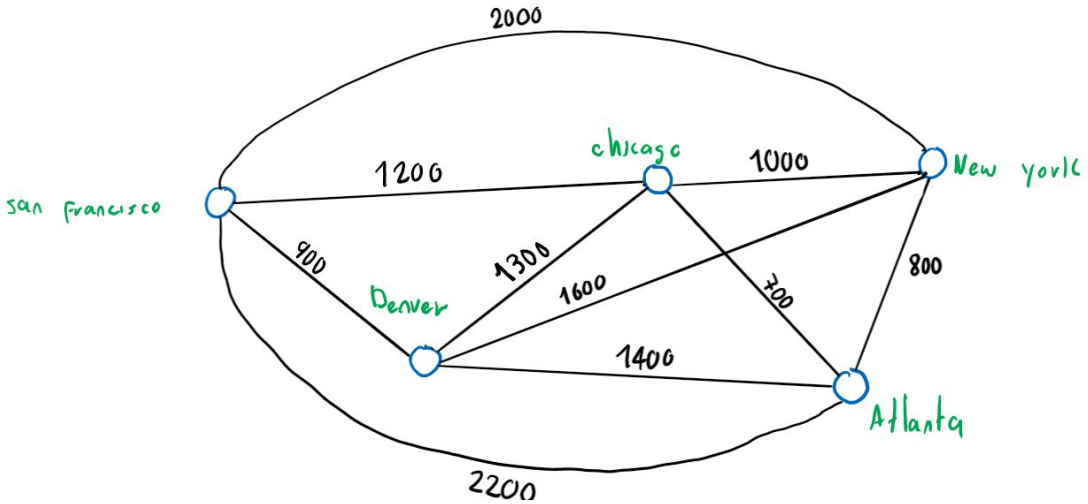
Test objective:

Check if the dfs method returns a minimum cost tree turned into a hashtable with all the previous vertices of the current existing vertices of the graph.

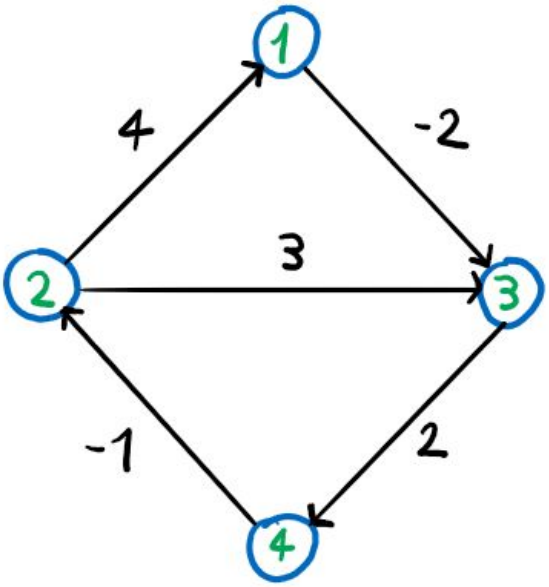
Class	Method	Scenario	Input values	Result
Graph	dfs	setup8	u	A hashtable with the previous vertices of the existing vertices. We verify the previous of each vertex in the hashtable. There is a visual of the result of the method in the annexed 2.5

Anexes:

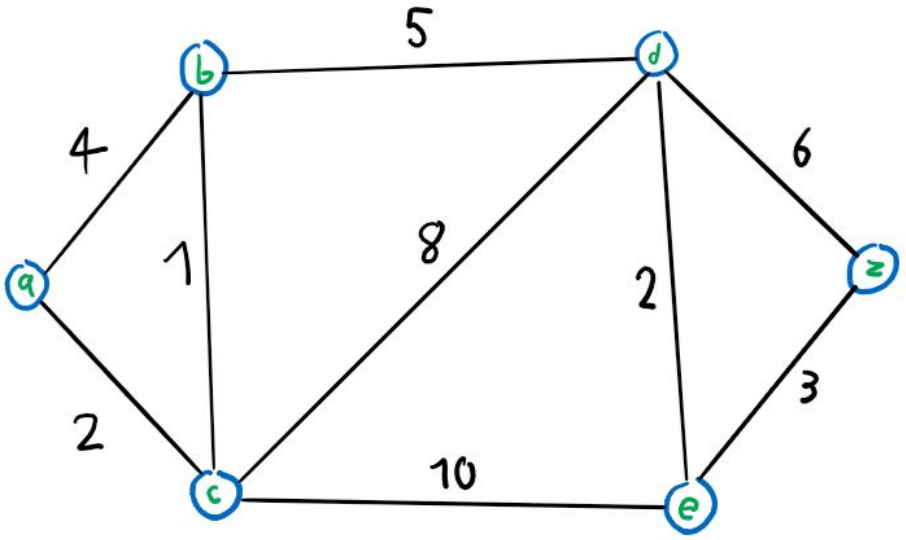
setups:
1.1



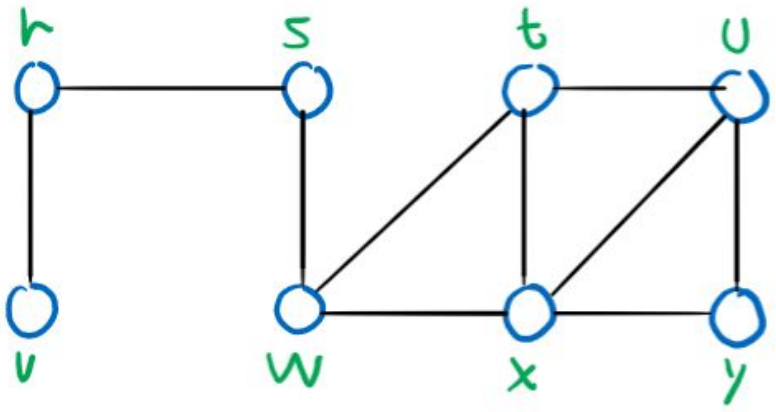
1.2



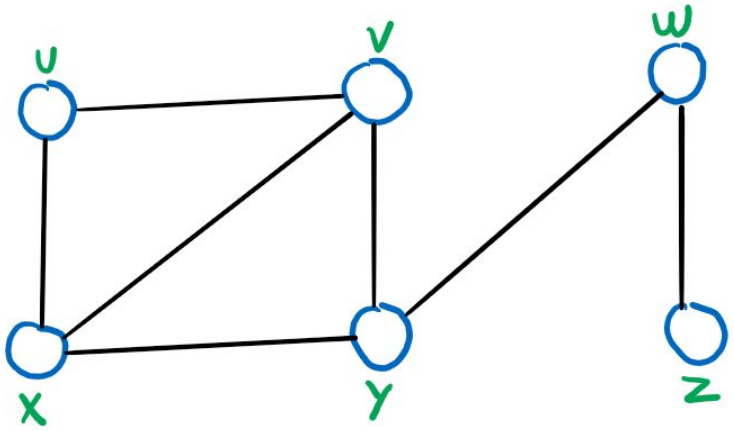
1.3



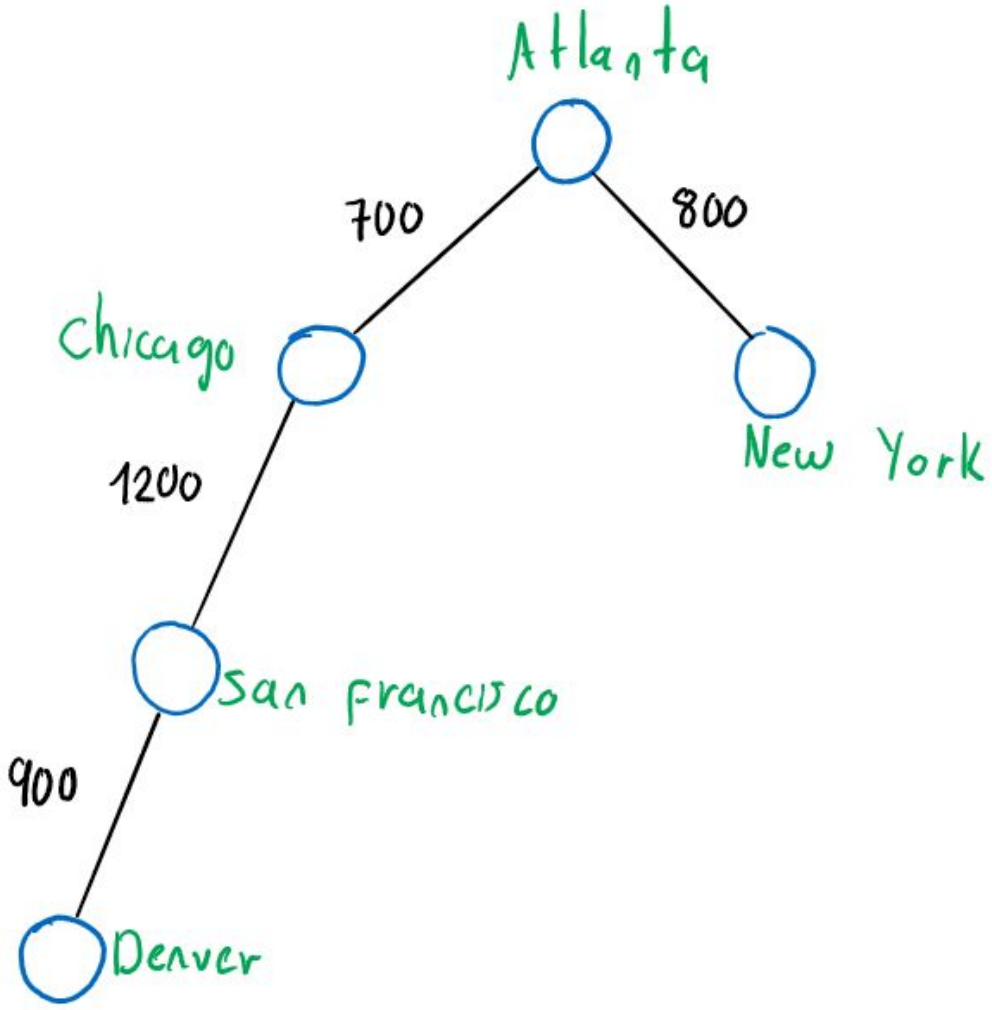
1.4



1.5



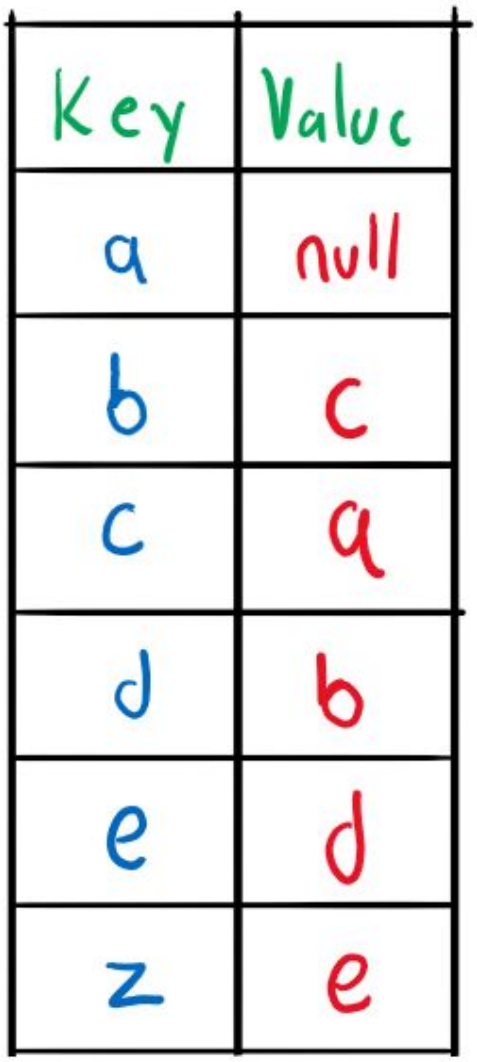
Test cases:
2.1



2.2

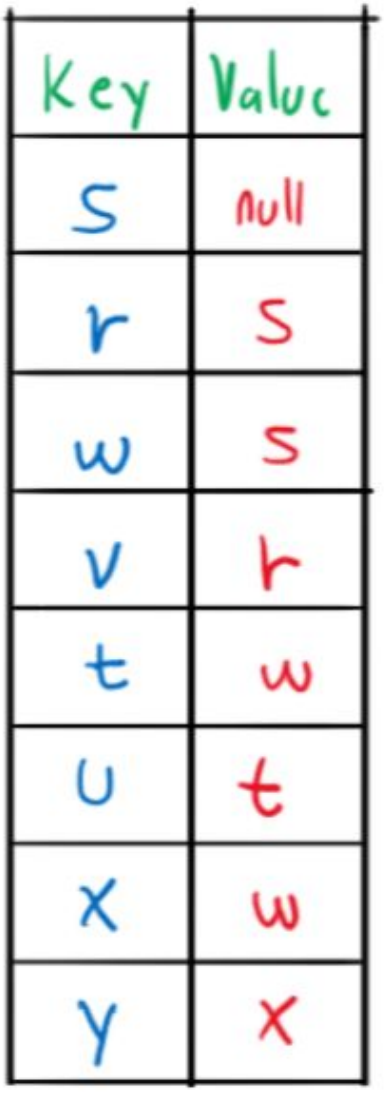
k = 4	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

2.3



Key	Valuc
a	null
b	c
c	a
d	b
e	d
z	e

2.4



Key	Value
s	null
r	s
w	s
v	r
t	w
u	t
x	w
y	x

2.5

Key	Value
u	null
v	u
x	y
y	v
w	y
z	w