**INTEGRATIVE TASK II**

**ENGINEERING DESIGN METHOD**

***Developed by:***   
  
*Vanessa Sánchez Morales (A00397949)*

*Luis Manuel Rojas Correa (A00399289)*

*Gabriel Escobar (A00399291)*

Icesi University

Professor: Dr. Marlon Gomez Victoria

Santiago de Cali,

Republic of Colombia

November 3rd, 2023

**Marlon Mania**

For our second integrative task we have been asked to develop a game. This game must be implemented using graphs. To accomplish this task, we have decided to develop Marlon Mania, a single player a game.

In this game players will have to go deep into the sewage system to solve a massive problem caused by a huge earthquake. This earthquake has destroyed some crucial parts of the sewage system leaving most of the population without access to water. To solve this problem the player must be able to connect two sets of pipes that were disconnected due to the earthquake. The starting point will be known as the source and the arrival point will be known as the drainage.

The idea is that the player achieves this connection using the least amount of pipes and for the most serious damage, the player must place the pipes so that the water takes the shortest time to arrive. In order to do this, the player is able to place 3 different types of pipes: the first one is the horizontal pipe that allows water to flow form right to left or vice versa, the second one is the vertical pipe that allows water to flow up and down or vice versa, and finally the third one is the circular pipe, which allows to change the flow of water from up or down to left or right. Lastly, players must know that the game calculates the score based on how effective the player's solution is.

<https://drive.google.com/file/d/1pLxAWU6AByXCs3OCBiwve3UbdZzsG4Bx/view>

**ENGINEERING DESIGN METHOD**

**PHASE 1: IDENTIFICATION OF THE PROBLEM (Software Requirement Specification-SRS)**

|  |  |
| --- | --- |
| **REQUIREMENT ANALYSIS TABLE** | |
| **Client** | Dr. Marlon Gómez Victoria |
| **User** | Players |
| **Functional Requirements** | * RF1: Start a new game. * RF2: View Top scores. * RF3: Place pipeline * RF4: Verify sewer system |
| **Problem Context** | *Marlon Mania game consists of a sewer system simulation.*  *In this game, the player can locate three different types of “pipes” within an 8x8 board, with the objective of connecting the “water source” to the “draining pipe” in the most efficient way. The users can also view a best score ranking.* |
| **Non-Functional Requirements** | * The program must two different graphs implementations. * The program must use at least two different graph algorithms. * The program must be developed with a user interface. * The project must use a version control software such as git. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier and Name** | ***RF1: Start a new game*** | | |
| **Summary** | *To start a new game the player must give his nickname and choose the difficulty level.*  *1.Easy (User must connect source and drain using the least amount of feet) 2. Difficult (User must connect source and drain ensuring water moves from one point to another in the shortest amount of time).*  *Then, the system will display an 8x8 board with an “S” and a “D”, each one representing the water source and the draining pipe respectively. Besides an option menu to play the game will be shown.* | | |
| **Input** | **Input name** | **Data type** | **Valid condition** |
| **User Nickname** | **String** | ***Can´t be empty*** |
| **Game Difficulty** | Int | *1.Easy*  *2.Hard* |
| **Result or Postcondition** | After the system receives the data entered by the user, it will show the game options (place pipe, verify and exit ) and the 8x8 board game, randomly locating an “F” and a “D”, each one representing the water source and the draining pipe respectively. It also takes the time in which the player started the game to calculate its game time. | | |
| **Output** | **Output name** | **Data type** | **Format** |
| gameBoard | Graph | X  X  X  X  X  X  X  X  X  X  X  X  X  X  **S**  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  **D**  X  X  X  X  X  X  X  X  X  X  X  X  X  X |
| gamePlayMenu | String | *1.Place a pipe*  *2.Verify game*  *3.Return (Exit)* |

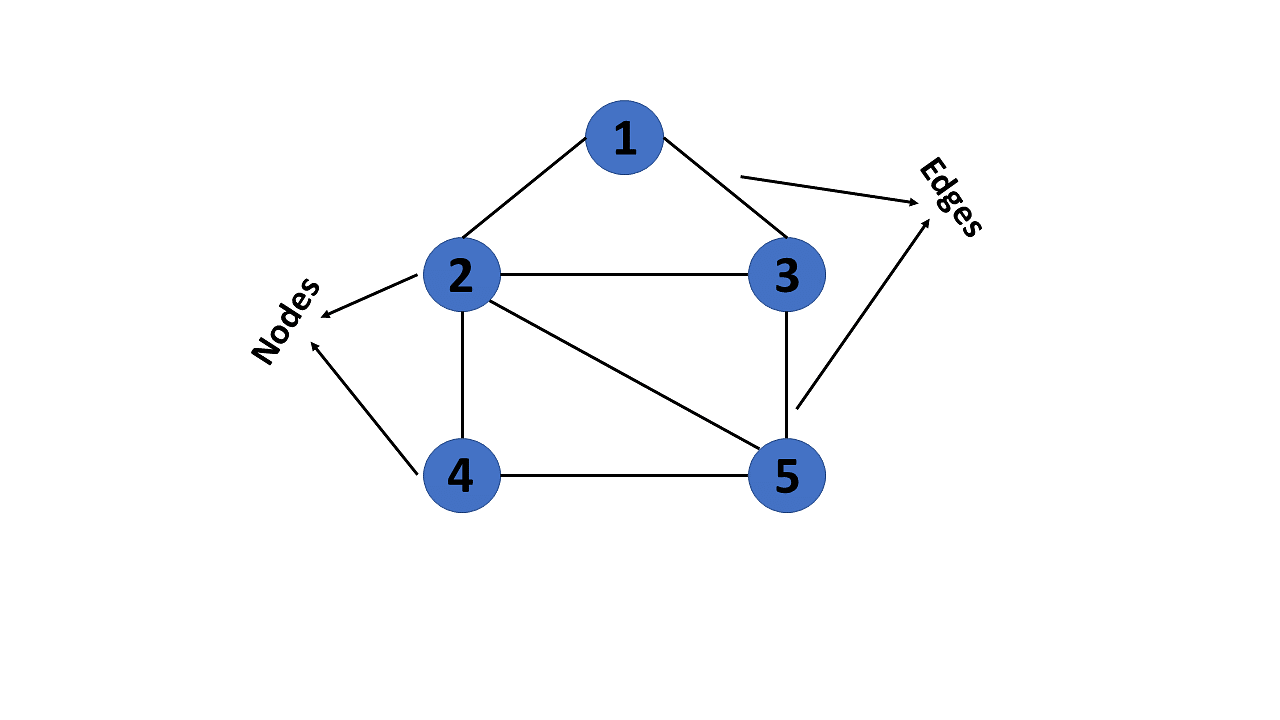
|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier and Name** | ***RF2: View scores*** | | |
| **Summary** | *The system must show, in descending order, the final scores of the players that have played and finished a game. This will happen if the user chooses the “View scores” option (2) in the menu.* | | |
| **Input** | **Input name** | **Data type** | **Valid condition** |
| **User Nickname** | **String** | ***Can´t be empty*** |
| N/A |  |  |
| **Result or Postcondition** | After each successful game, the system calculates the user’s score. The system will calculate the player´s final score using the following formulas accordingly to the game difficulty:  1)Score= |(1000)\*( optimal number of pipes / number of pipes used) |  2)Score= | (1000) \*(optimal time of water flow/ time of the pipes used ) |  The time each player takes to play the game will be calculated during the game. | | |
| **Output** | **Output name** | **Data type** | **Format** |
| Final Score List |  | *1)PlayerNickname = Score* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier and Name** | ***RF3: Place Pipes*** | | |
| **Summary** | *After entering the option to Start a new game menu (see RF1), The system must allow the user to locate a “pipe” in a specific position of the 8x8 board, by asking for the coordinates in which the new pipe will be located and the type of pipe.* | | |
| **Input** | **Input name** | **Data type** | **Valid condition** |
| xCoordinate | int | Must be an Integer [1-8] |
| yCoordinate | int | Must be an Integer [1-8] |
| pipeType | int | Must be an Integer [1-3]  1.Horizontal ( = )  2.Vertical ( || )  3.Circular( o ) |
| **Result or Postcondition** | The system searches for the coordinate that the player chose. If said coordinate is located within the possible range of the board and it isn´t occupied by an “S” or “D”, the type of pipe that the player chose will be displayed in the [x,y] coordinate of the board. Else, the board will appear with no changes made. | | |
| **Output** | **Output name** | **Data type** | **Format** |
| gameBoard | graph | *8x8 board of “x” characters, contains an S (water source), a D (draining pipe) and the pipes (“=”,”| |”,”o”)* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier and Name** | ***RF4: Verify sewer system*** | | |
| **Summary** | *The system must verify that the pipes’ solution provided by the user is valid, meaning, the water source (“S”) is connected to the draining pipe (“D”) with a correct usage of the pipes (“=”,”||”,”o”). Then, the user will see if their option is correct or not, and depending on that, the game will close.* | | |
| **Input** | **Input name** | **Data type** | **Valid condition** |
| N/A | N/A | N/A |
| **Result or Postcondition** | The system checks that the pipes are located correctly according to their type and direction. The “S” and “D” must be connected, allowing the water flow, with the “=” pipes going one next to the other, and the “|  |” pipes going one under the other. Besides, an “o” cannot be next to another “o” or to the “S” and “D”, and it can only be used to do a 90° spin with the pipes.  If the solution is correct, the game is closed, the time is taken, calculated and the score saved; else, the game continues, and the menu and board will be displayed again. | | |
| **Output** | **Output name** | **Data type** | **Format** |
| message | String | *Whether the user´s option was correct or not. It can be:*  *“The solution is correct”*  *“The sewer system is not correct”* |

**FASE 2: COMPILATION OF NECESSARY INFORMATION**

**Important Terminology**

Graph:

Graphs in data structures are non-linear data structures made up of a finite set of nodes or vertices and the edges that connect them. Graphs in data structures are used to address real-world problems in which it represents the problem area as a network. The graph is denoted by G(E, V).

* Vertex:

The vertices are the fundamental units of the graph, sometimes also denominated as node. Every node/vertex can be labeled or unlabeled.

* Edge:

Edges connect two nodes of the graph in any possible way. Depending on the type of graph, the set of vertices it connects must be ordered or not. Sometimes known as arcs, they can be labeled/unlabeled (also determined as weight of the edge).

* Path:

A path (of length n) in an (undirected) graph G is a sequence of vertices {v0, v1, ..., vn-1, vn} such that there is an edge between vi and vi+1 ∀i ∈ [0..n-1] along the path.

* Adjacency:

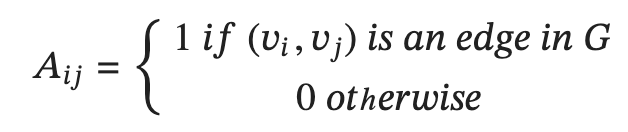
A vertex is said to be adjacent to another vertex if there is an edge in common connecting them.

Adjacency List:

An adjacency List (AL) is an array of V lists, one for each vertex (usually in increasing vertex number) where for each vertex i, AL[i] stores the list of i's neighbors or adjacent nodes. For weighted graphs, the list can store pairs of (neighbor vertex number, weight of this edge) instead.

Adjacency Matrix:

The adjacency matrix of a simple labeled graph G is the matrix *A* with A[[*i,j*]]=1 or 0 according to whether the vertex *vj*, is adjacent to the vertex *v*j or not. For simple graphs without self-loops, the adjacency matrix has 0 s on the diagonal. For undirected graphs, the adjacency matrix is symmetric.



User Interface:

User Interface (UI) is the point at which the human interacts with a computer, website and/or application. The UI must be intuitive and ease the user’s experience with the usage of the software product, requiring minimum effort on the user's part to receive the maximum desired outcome.

Dijkstra’s Algorithm:

Algorithm for finding the shortest path from a starting node to a target node in a weighted graph. The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.

Minimum Spanning Tree:

A spanning tree of a graph is a collection of connected edges that include every vertex in the graph, but that do not form a cycle. The Minimum Spanning Tree is the one whose cumulative edge weights have the smallest value possible.

Pipe:

Connecting lines or tubes between (for the context of the program) other pipes. They act as edges between two of the vertices of the graph that represent the matrix of 8x8 that will be the playground for each of the user’s game.

They are classified in 4 different types (for more characterization of each one of them, look at phase 1: SRS).

* Default (x)
* Vertical ( | | )
* Horizontal (=)
* Circular ( ° )

**PHASE 3: RESEARCH OF CREATIVE SOLUTIONS**

**Alternative 1**

1. **User Interface:**

JavaFX is a modern library to create user interaction interfaces and can be used in various platforms such as Windows, macOS and Linux. Interactive Graphics can be created with this library.

1. **Game:**

Marlon Mania is a game that consists of a sewer system simulation. In this game, the player can locate three different types of “pipes” within an 8x8 board/matrix, with the objective of connecting the “water source” to the “draining pipe”. The users can also view a players’ ranking according to the scores gained, with each of the players’ names. There are two levels of difficulty: easy, where the player must complete the connect the water source to the draining pipe; and hard, where the player, to get points, must connect the last-mentioned pipes by using the shortest path.

1. **Versions of graph:**

For the first version of the graph, an adjacency list is considered.

An adjacency List (AL) is a representation of a graph through an array of V lists, one for each vertex (usually in increasing vertex number) where for each vertex i, AL[i] stores the list of adjacent nodes. For weighted graphs, the list can store pairs of (neighbor vertex number, weight of this edge) instead.

For the second version of the graph, an adjacency matrix is considered.

The adjacency matrix of a simple labeled graph G is the matrix A with A[[i,j]]=1 or 0 according to whether the vertex vj, is adjacent to the vertex vj or not.

1. **Graph Algorithms:**

For the first graph algorithm to be used, a graph-traversal algorithm is considered, specifically DFS.

Depth-first search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored.

For the second graph algorithm to be used, a Single-Source Shortest Path (SSSP) algorithm (given a source vertex it finds the shortest path from the source to all other vertices) is considered, specifically Dijkstra’s algorithm.

Dijkstra's Algorithm works on the basis that, in a Graph G={A,B,C,D}, any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D. Dijkstra used this property in the opposite direction overestimating the distance of each vertex from the starting vertex. Then each node and its neighbors are visited to find the shortest subpath to those neighbors.

**Alternative 2**

1. **User Interface:**

It will be created using the GUI package, composed of the two classes AWT and Swing, which are rich in components and containers (Component, container, Jcomponent, Jframe, Jdialog, JApplet, Jpanel, Graphics), whose tools and classes are interactable and simple, making the user interface creating process more practical.

1. **Game:**

Pipe-Mania is a game that consists of a sewer system simulation. In this game, the player can locate three different types of “pipes” within an 8x8 board/matrix, with the objective of connecting the “water source” to the “draining pipe”. The users can also view a players’ ranking according to the scores gained, with each of the players’ names.

The player, to get points, must connect the water source to the draining pipe.

1. **Versions of Graph:**

For the first version of a graph, an incidence matrix is considered.

The incidence matrix A of an undirected graph has a row for each vertex and a column for each edge of the graph. The element A[[i,j]]=1 if the ith vertex is a vertex of the jth edge, otherwise A[[i,j]]=0.

For the second version of a graph, an edge list is considered.

An edge list is a list or array of all the edges in a graph. The underlying data structure for keeping track of all the nodes and edges is a single list of pairs. Each pair represents a single edge and is comprised of the two unique IDs of the nodes involved. Each line/edge in the graph gets an entry in the edge list, and that single data structure then encodes all nodes and relationships.

1. **Graph Algorithms:**

For the first graph algorithm to be used, a graph-traversal algorithm is considered, specifically BFS.

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

For the second graph algorithm to be used, a Shortest Path (SP) algorithm (finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized) is considered, specifically Floyd-Warshall’s algorithm.

It computes the computes the shortest distances between every pair (or a given pair) of vertices in the input graph.

**Alternative 3**

1. **User Interface:**

It will be created by console, making the effort of using different java functionalities and colors to make the user experience more user friendly, whilst making it easier for the programmer to implement their data structures, logic and functionality of the program with more flexibility.

1. **Game:**

Pipe-Mania is a game that consists of a sewer system simulation. In this game, the player can locate three different types of “pipes” within an 6x6 board/matrix, with the objective of connecting the “water source” to the “draining pipe”.

The player, to get points, must connect the water source to the draining pipe.

1. **Versions of Graph:**

For the first version of a graph, an edge list is considered.

An edge list is a list or array of all the edges in a graph. The underlying data structure for keeping track of all the nodes and edges is a single list of pairs. Each pair represents a single edge and is comprised of the two unique IDs of the nodes involved. Each line/edge in the graph gets an entry in the edge list, and that single data structure then encodes all nodes and relationships.

For the second version of a graph, an adjacency list is considered.

An adjacency List (AL) is a representation of a graph through an array of V lists, one for each vertex (usually in increasing vertex number) where for each vertex i, AL[i] stores the list of adjacent nodes. For weighted graphs, the list can store pairs of (neighbor vertex number, weight of this edge) instead.

1. **Graph Algorithms:**

For the graph algorithms, an alternative is to use MST (Minimum Spanning Tree) algorithms.

In the first place is Kruskal’s algorithm. In Kruskal’s algorithm, all edges of the given graph are sorted in increasing order. Then new edges and nodes keep being added in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last.

For the second algorithm, Prim’s is an alternative.

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

**PHASE 4: TRANSITION FROM IDEA FORMULATION TO PRELIMINARY DESIGNS**

**Alternative 1 (Chosen):**

1. **User Interface:**

* JavaFx allows the use of more animation and visual functionalities.
* It requires more time and effort to learn how to use said library.
* JavaFx usage applies the model, control, view design pattern that guarantees the application of correct programming practices.

1. **Game:**

* Marlon Mania´s game needs the usage of graphs to verify the route of pipes made by the player
* It is an entertaining game that requires design principles and algorithmic logic to be developed, following, that way, with the principles of said integrative task.
* The game, by having an 8x8 board, accomplishes the non-functional requirement of using a graph with more than 50 vertices (as it has 64 nodes).
* It has different levels and game complexities that not only enrich the gaming experience, but also apply different graph algorithms that are mandatory (non-functional requirement).

1. **Versions of Graph:**

* An adjacency list clearly states the relations between the vertices and the edges that those relationships form. It also facilitates the management of the vertices, where each node is managed as an object, which will have control and information of the edges formed with itself.
* An adjacency matrix clearly states the relationships between each pair of vertices. Although its methods and the structure itself entail a bigger spatial complexity, the management of the vertices and edges, as they are described by 1 and 0´s in the case of the non-weighted graphs, and in contrast with other structures, it also eases the visualization and storage of weighted graphs.

1. **Graph Algorithm:**

* DFS is an algorithm that can be easily implemented and adapted to the developer’s and the project’s necessities. It also solves one of the game’s functionalities, which is verifying whether the water flow is correctly built. In other words, it efficiently checks whether two nodes or the nodes of a graph are connected.
* Although Dijkstra's Algorithm algorithmic complexity is significant, it solves a NP-Complete problem and solves one of the functionalities of the game, which is finding the shortest/less weighted path in a weighted graph.

**Alternative 2 (Chosen):**

1. **User Interface:**

* The GUI package can be easily implemented in the development of a Java application.
* The package and its containers (Component, container, Jcomponent, Jframe, Jdialog, JApplet, Jpanel, Graphics) are practical when used and accomplish the development of a user-friendly application.

1. **Game:**

* Pipe Mania´s game needs the usage of graphs to verify the route of pipes made by the player
* It is an entertaining game that requires design principles and algorithmic logic to be developed, following, that way, with the principles of said integrative task.
* The game, by having an 8x8 board, accomplishes the non-functional requirement of using a graph with more than 50 vertices (as it has 64 nodes).

1. **Versions of Graph:**

* An incidence matrix states the relationships between a vertex and a specific edge. Although its methods and the structure itself entail a bigger spatial complexity, the management of the vertices and edges, as they are described by 1 and 0´s.
* An edge list contains the information of all the relationships between each of the vertices of the graph (the edges). Although its spatial complexity is smaller than other versions of the graph, the methods that it entails require bigger algorithmic time and the vertices are not managed by themselves but only as pairs.

1. **Graph Algorithm:**

* BFS is an algorithm that can be easily implemented. It also solves one of the game’s functionalities, which is verifying whether the water flow is correctly built. In other words, it efficiently checks whether two specific and/or given vertices or the nodes of a graph are connected.
* Floyd-Warshall’s algorithm, like Dijkstra’s, solves a NP-Complete problem and solves one of the functionalities of the game, which is finding the shortest/less weighted path in a weighted graph. However, their differences rely on the fact that Floyd-Warshall’s returns the shortest path (not checks) and has bigger algorithmic complexity.

**Alternative 3 (Discarded):**

1. **User Interface:**

* Although an application that runs by console is easier to implement and is more flexible to changes, it lacks design, doesn´t require the application of design patterns and is not user-friendly. Meaning, it doesn’t satisfy the non-functional requirement of the project of having an user interface and it affects the gaming experience.
* Therefore, this option is completely discarded.

1. **Game:**

* The version of Pipe Mania described in this alternative doesn’t follow many of the requirements of the projects.
* Firstly, by having a 6x6 game board, it does not satisfy the non-functional requirement of the project of implementing a graph with more than 50 nodes/vertices.
* Secondly, by not counting with any levels of difficulty, this solution does not require the implementation of different graph algorithms, which is also mandatory in the development of the game.
* Finally, this version of the game does not count with any levels of difficulty or points ranking that motivates the player or enrich their gaming experience.
* Therefore, this option is discarded.

1. **Versions of Graph:**

* An edge list contains the information of all the relationships between each of the vertices of the graph (the edges). Although its spatial complexity is smaller than other versions of the graph, the methods that it entails require bigger algorithmic time and the vertices are not managed by themselves but only as pairs.
* An adjacency list clearly states the relations between the vertices and the edges that those relationships form. It also facilitates the management of the vertices, where each node is managed as an object, which will have control and information of the edges formed with itself.
* Although these versions of graph are possible candidates for their implementation, it has already been proposed in other alternatives and does not bring anything new to the brainstorming process.

1. **Graph Algorithm:**

* MST (Minimum Spanning Tree), although they can be helpful in various applications, they do not achieve the purpouse of this project, which is the one of checking the connection between vertices of a graph and finding the less weighted path.
* In conclusion, this option is discarded.

PHASE 5: EVALUATION AND SELECTION OF THE SOLUTION

***5.1 Criteria Evaluation Definition in terms of Quality:***

|  |  |  |
| --- | --- | --- |
| **CRITERIA QUALITY EVALUATION DEFINITION** | | |
| **Criteria** | **Definition** | **Evaluation Scale** |
| Functionality | Assess how well the software meets the required functionality, including precision, adequacy, interoperability, conformance, and security. | -1: Does not meet functionality requirements  -2: Partially meets functionality requirements  -3: Moderately meets functionality requirements  -4: Adequately meets functionality requirements  -5: Fully and precisely meets functionality requirements |
| Reliability | Evaluate the software's reliability in terms of maturity, error tolerance, and recoverability. | -1: Highly unreliable and lacks maturity  -2: Moderately reliable with some maturity  -3: Reasonably reliable with good maturity  -4: Highly reliable and mature  -5: Exceptionally reliable and mature |
| Usability | Assess the usability of the software, including comprehensibility, learnability, operability, and attractiveness. | -1: Highly unusable, poor user experience  -2: Moderately usable but needs improvement  -3: Reasonably usable with a good user experience  -4: Highly usable with an excellent user experience  -5: Exceptionally usable and provides an outstanding user experience |
| Efficiency | Evaluate the software's efficiency in terms of response time (algorithmic complexity), memory usage, and resource utilization. | -1: Highly inefficient and consumes excessive resources  -2: Moderately efficient but could be more resource-friendly  -3: Reasonably efficient with acceptable resource usage  -4: Highly efficient with minimal resource consumption  -5: Exceptionally efficient, making optimal use of resources |

***5.1.2 Evaluation of the Chosen Alternatives in terms of Quality:***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ALTERNATIVE 1 EVALUATION** | | | | |
| **Requisit** | **Functionality Punctuation** | **Reliability**  **Punctuation** | **Usability**  **Punctuation** | **Efficiency**  **Punctuation** |
| 1. *User Interface* | 5 | 4 | 4 | 5 |
| 1. *Game* | 5 | 5 | 5 | 4 |
| 1. *Versions of Graph* | 5 | 5 | 5 | 5 (adjacency list)  +  3 (adjacency matrix) |
| 1. *Graph Algorithms* | 5 | 5 | 5 | 4 (DFS)  +  4 (Dijkstra’s) |
| ***Total Punctuation*** | **20** | **19** | **19** | **25** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ALTERNATIVE 2 EVALUATION** | | | | |
| **Requisit** | **Functionality Punctuation** | **Reliability**  **Punctuation** | **Usability**  **Punctuation** | **Efficiency**  **Punctuation** |
| 1. *User Interface* | 3 | 3 | 3 | 3 |
| 1. *Game* | 3 | 4 | 4 | 4 |
| 1. *Versions of Graph* | 3 | 4 | 2 | 3 (incidence matrix)  +  2 (edge list) |
| 1. *Graph Algorithms* | 2 | 4 | 4 | 4 (BFS)  +  1 (Floyd-Warshall’s) |
| ***Total Punctuation*** | **11** | **15** | **13** | **17** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Criteria** | Functionality | Reliability | Usability | Efficiency | **Total** |
| **Alternative 1** | **20** | **19** | **19** | **25** | **83** |
| **Alternative 3** | **11** | **15** | **13** | **17** | **56** |

***5.2 Criteria Evaluation Definition in terms of Algorithm and Spatial Complexity (Worst Case):***

****

Font: Medium

[*https://medium.com/@Hollyzhou/data-structure-the-big-o-notation-e3e2405bb8eb*](https://medium.com/@Hollyzhou/data-structure-the-big-o-notation-e3e2405bb8eb)

|  |  |  |
| --- | --- | --- |
| **BIG (O) EVALUATION SCALE** | | |
| **Complexity** | **Definition** | **Punctuation** |
| Cubic |  | 0 |
| Quadratic |  | 1 |
| Lineal | O(n) | 2 |
| Log-linear Complexity | O(n Log(n)) | 3 |
| Logarithmic | O(Log(n)) | 4 |
| Constant | O(1) | 5 |

|  |  |  |
| --- | --- | --- |
| **ALGORITHMICAL EVALUATION CRITERIA** | | |
| **Criteria** | **Definition** | **Evaluation Scale** |
| **Verifying sewer system** | | |
| *Checking the water flow for Easy Level* | This criterion evaluates how efficiently the system checks that the pipes are correctly connected, meaning, that the vertices are connected.  For this functionality, the DFS or BFS algorithm would be used | BIG O CRITERIA |
| *Checking the water flow for the Hard Level* | This criterion evaluates how efficiently the system checks that the pipes are correctly connected, meaning, that the vertices are connected. Besides, it also guarantees that the path built is the most efficient one.  For this functionality, Dijkstra’s or Floyd-Warshall’s algorithm would be used. | BIG O CRITERIA |

***5.2.1 Evaluation of Structures in Chosen Alternatives in terms of Algorithm Complexity***

***(Worst Case):***

**Alternative 1 Evaluation**

|  |  |
| --- | --- |
| **ALTERNATIVE 1 (DFS & Dijkstra’s Algorithm)**  **Verifying sewer system** | |
| **Criteria** | **Evaluation Scale**  **(Big-O)** |
| *Checking the water flow for Easy Level* | 2  [O(V+E)] |
| *Checking the water flow for the Hard Level* | 1 |
| ***Total Punctuation*** | **3** |

**Alternative 3 Evaluation**

|  |  |
| --- | --- |
| **ALTERNATIVE 1 (BFS & Floyd-Warshall’s Algorithm)**  **Verifying sewer system** | |
| **Criteria** | **Evaluation Scale**  **(Big-O)** |
| *Checking the water flow for Easy Level* | 2  [O(V+E)] |
| *Checking the water flow for the Hard Level* | 0 |
| ***Total Punctuation*** | **2** |

**Comparison of Alternative 1 and Alternative 3 Algorithmical Time Complexity:**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Verifying Sewer System** | |
| *Checking the water flow for Easy Level* | *Checking the water flow for the Hard Level* |
| **Alternative 1** | **2** | **2** |
| **Alternative 3** | **1** | **0** |
| **Total** | **3** | **2** |

***5.3 Selection of the best Alternative:***

Alternative 1:

This solution has been selected due to its comprehensive approach to the task at hand. Firstly, it is possible to observe that its quality fit properly not only the objectives of the game proposed, but also the requisites of the integrative task. On the other hand, the algorithms proposed by alternative one, and that will deal with the functional requirement of verifying the sewer system, are more effective than the second alternative and embrace more integrally the objectives of the levels of the game proposed.

PHASE 6: PREPARATION OF REPORTS AND SPECIFICATIONS

***6.1 General Problem Specification:***

Marlon Mania is a single player a game in which players will have to go deep into the sewage system to solve a massive problem caused by a huge earthquake. This earthquake has destroyed some crucial parts of the sewage system leaving most of the population without access to water. To solve this problem the player must be able to connect two sets of pipes that were disconnected due to the earthquake. The starting point will be known as the source and the arrival point will be known as the drainage.

The idea is that the player achieves this connection using the least amount of pipes and for the most serious damage, the player must place the pipes so that the water takes the shortest time to arrive. In order to do this, the player is able to place 3 different types of pipes: the first one is the horizontal pipe that allows water to flow form right to left or vice versa, the second one is the vertical pipe that allows water to flow up and down or vice versa, and finally the third one is the circular pipe, which allows to change the flow of water from up or down to left or right. Lastly, players must know that the game calculates the score based on how effective the player's solution is.

To model the situation at hand, a graph will be used as the main data structure. The user must have the possibility of switching between two different versions of graph (Adjacency List and Adjacency Matrix) and to check whether the user’s solution is correctly built, two different types of graph’s algorithms will be implemented (DFS and Dijkstra’s Algorithm).

***6.2 SubProblems Specification:***

|  |  |
| --- | --- |
| **Storing Tasks and Reminders** | |
| *Sub-Problem Specification* | The program has to store in a hash table the tasks and reminders established by the user; with its own identifier, title, description, due date and priority. |
| *Inputs* | * **Inp\_1:**Task or Reminder Identifier * **Inp\_2:**Task or Reminder Title * **Inp\_3:** Task or Reminder Description * **Inp\_4:** Task or Reminder due Date * **Inp\_5:** Task or Reminder Priority |
| *Outpust* | * **Out\_1:** Confirmation Message. Example Gratie: “*Successfully Stored”* |
| ***Considerations:***   * After the system receives the data entered by the user, it creates a new task, with its state as “UNDONE”. It uses the hash function to create the key of said task (for it to be accessed later). If the task is a “PRIORITY” it is saved in a Stack structure; otherwise, it will be saved in a Queue structure. The action is saved in the list of actions. * All inputs cannot be stored empty * Due date has to be in a future date according to the current date of task-reminder creation | |

|  |  |
| --- | --- |
| **User Interface** | |
| *Sub-Problem Specification* | A comprehensible, useful and practical interface has to be shown to the user. Where he can add, modify and delete new or existing tasks and reminders. Also, the interface has to organize the activities according to their importance or due date. |
| *Inputs* | * **Inp\_1:** Option (*“Add new Task”*) * **Inp\_2:** Option (“Modify Task) * **Inp\_3:** Option (“Delete Task “) * **Inp\_4:** Option (“Organize Task by Priority) * **Inp\_5:** Option (“Organize Task by due Date) |
| *Outpust* | * **Out\_1:** List of tasks and reminders * **Out\_2:** List of tasks and reminders updated |
| ***Considerations:***   * After the system receives the data entered by the user, it searches the task with the key entered. If the task is found, the attribute chosen is modified according to the data entered by the user; otherwise, the user will be indicated that the task was not found. The action is saved in the list of actions. * After the system receives the data entered by the user, it searches for the task with the key entered. If the task is found, it is deleted from both the list of tasks and the pile of the priority where it belonged; otherwise, the user is indicated that the task wasn’t found. The action is saved in the list of actions. * After the system receives the data entered by the user, it searches the task with the key entered. If the task is found, its information is shown in the interface to the user; otherwise, the user is indicated that the task wasn’t found. The action is saved in the list of actions. | |

|  |  |
| --- | --- |
| **Priorities Management** | |
| *Sub-Problem Specification* | There are two subcategories of tasks and reminders: priorities and non priorities:   * Prioritary Taks: Has to be stored in a queue according to their importance level; most important tasks have to be shown first. * Non-priority tasks: They have to be stored in a stack, where the user can manage them according to their arrival order. |
| *Inputs* |  |
| *Outputs* | * Out\_1: Tasks stored and sorted according to their importance level. |
| ***Considerations:***  -The tasks and priorities are shown to the user interface, according to their level of importance. If there are no tasks or reminders in both categories it has to show a message of emptiness, | |

|  |  |
| --- | --- |
| **Undoing Actions Methods** | |
| *Sub-Problem Specification* | To implement an "Undo" function, the system utilizes a Last-In-First-Out (LIFO) stack to track user actions, encompassing tasks' addition, modification, and deletion. Each stack entry comprehensively records action specifics and associated task information. When a user executes an action, it is promptly documented in the stack. The system incorporates an "Undo" method, affording users the capability to reverse their latest action by extracting the most recent entry from the stack and subsequently undoing the corresponding action based on the logged data. This user-oriented "Undo" feature, available within the interface, significantly enhances the system's usability by providing seamless error correction. |
| *Inputs* | * **Inp\_1:** Undo action |
| *Outputs* | * **Out\_1:** The action is reversed to the previous stored status |
| ***Considerations:***   * If there are no previous states, the action does not change | |

**PHASE 7: Design Implementation**

**Class Diagram:**

[Task manager.pdf](https://drive.google.com/file/d/1EaBc4z791w61DxUYIFtVaLxQWyP1voVJ/view?usp=sharing)

**Hash Table Task to Implement:**

**HashEntry**

|  |  |
| --- | --- |
| **HashEntry Subroutines** | |
| **Subroutine Specification** | **Construction** |
| |  |  | | --- | --- | | **Name:** | HashEntry () | | **Description** | It instantiates a new node that will be stored in the hash table. | | **Input** | <K> key <V> value | | **Return** | none  (Object of Hash Entry instantiated) | | **public HashEntry(K key, V value) {**  **this.key = key;**  **this.value = value;**  **this.next=null;**  **this.prev=null;**  **}** |

|  |  |
| --- | --- |
| **Hash Table Subroutines** | |
| **Subroutine Specification** | **Construction** |
| |  |  | | --- | --- | | **Name:** | hashTable() | | **Description** | It instantiates a new hash table with a default size of 10 | | **Input** | none | | **Return** | none  (Hash Table Object instantiated) | | **public HashTable(){**  **table = new HashEntry[DEFAULT\_SIZE];**  **this.existingNodes = 0 ;**  **}** |
| |  |  | | --- | --- | | **Name:** | hashFunction() | | **Description** | converts any type of key into the index where the element will be inserted in the hash table | | **Input** | <K> key | | **Return** | int index | | **public int hashFunction(K key){**  **int hashCode;**  **hashCode = key.hashCode();**  **return Math.abs(hashCode) % table.length;**  **}** |
| |  |  | | --- | --- | | **Name:** | add() | | **Description** | inserts a new element into a hash table index, managing collisions if needed | | **Input** | <K> key <V> value | | **Return** | void | | **public void add(K key, V value){**  **int index= hashFunction(key);**  **HashEntry<K,V> newEntry= new HashEntry<>(key, value);**  **HashEntry<K,V> current=table[index];**  **if(current==null){**  **table[index]=newEntry;**  **}else{**  **while(current.getNext()!=null){**  **current=current.getNext();**  **}**  **current.setNext(newEntry);**  **newEntry.setPrev(current);**  **newEntry.setNext(null);**  **}**  **this.existingNodes++;**  **}** |
| |  |  | | --- | --- | | **Name:** | getFirst () | | **Description** | returns the first element that is stored in an index of the hash table | | **Input** | <K> key | | **Return** | (NODE) HashEntry<K,V>: if the node is founded  null: if the node is not stored in the hash table | | **public HashEntry<K,V> getFirst(K key){**  **if(table==null){**  **return null;**  **}**  **int index= hashFunction(key);**  **return table[index];**  **}** |
| |  |  | | --- | --- | | **Name:** | getValue () | | **Description** | returns the value of the first element that is stored in an index of the hash table | | **Input** | <K> key | | **Return** | <V> value: if the node is founded  null: if the node is not stored in the hash table | | **public V getValue(K key){**  **if(table==null){**  **return null;**  **}**  **int index= hashFunction(key);**  **if(table[index].getValue()==null){**  **return null;**  **}**  **return table[index].getValue();**  **}** |
| |  |  | | --- | --- | | **Name:** | find () | | **Description** | returns a node stored in an index that has more than one more stored; managing collisions. | | **Input** | <K> key | | **Return** | (NODE) HashEntry<K,V>: if the node is founded  null: if the node is not stored in the hash table | | **public HashEntry<K,V> find(K key){**  **int index= hashFunction(key);**  **HashEntry<K,V> current=table[index];**  **while(current!=null){**  **if(current.getKey().equals(key)){**  **return current;**  **}**  **current=current.getNext();**  **}**  **return null;**  **}** |
| |  |  | | --- | --- | | **Name:** | findValue () | | **Description** | returns a the value of a node stored in an index that has more than one more stored; managing collisions. | | **Input** | <K> key | | **Return** | <V> value: if the node if founded  null: if the node is not stored in the hash table | | **public V findValue(K key){**  **int index= hashFunction(key);**  **HashEntry<K,V> current=table[index];**  **while(current!=null){**  **if(current.getKey().equals(key)){**  **return current.getValue();**  **}**  **current=current.getNext();**  **}**  **return null;**  **}** |
| |  |  | | --- | --- | | **Name:** | delete () | | **Description** | delete a node that is stored in the hash table, managing also collisions: when there is more than one node stored in an index | | **Input** | <K> key <V> value | | **Return** | void | | **public void delete(K key, V value) {**  **int index = hashFunction(key);**  **if(table[index]==null){**  **System.out.println("Node not found!");**  **}else{**  **HashEntry<K, V> current = table[index];**  **while (current != null) {**  **if (current.getKey().equals(key) && current.getValue().equals(value)) {**  **if (current.getPrev() != null) {**  **current.getPrev().setNext(current.getNext());**  **}**  **if (current.getNext() != null) {**  **current.getNext().setPrev(current.getPrev());**  **}**  **if (current == table[index]) {**  **table[index] = current.getNext();**  **}**  **current.setNext(null);**  **current.setPrev(null);**  **this.existingNodes --;**  **return;**  **}**  **current = current.getNext();**  **}**  **}**  **}** |
| |  |  | | --- | --- | | **Name:** | isEmpty () | | **Description** | indicated if the hash table has or not has nodes stored | | **Input** | none | | **Return** | boolean  True if the hash table has no nodes  False if the hash table has at least one node | | **public boolean isEmpty(){**  **return this.existingNodes == 0;**  **}** |
| |  |  | | --- | --- | | **Name:** | showTable () | | **Description** | It converts the information (value) of the elements into a string, and stores it into a StringBuilder. To then convert it into a single string chain | | **Input** | none | | **Return** | String: value of the elements stored in a string chain.  String “No Elements Stored” Message | | **public String showTable(){**  **StringBuilder elements = new StringBuilder();**  **for(int i=0;i< table.length;i++) {**  **if (table[i] != null) {**  **elements.append("\t").append(table[i].getValue().toString()).append("\n");**  **HashEntry<K,V> current = table[i].getNext();**  **while (current != null) {**  **elements.append("\t").append(current.getValue().toString()).append("\n");**  **current = current.getNext();**  **}**  **}**  **}**  **if(elements.toString().isEmpty()){**  **return """**  **\t╔════════════════════╗**  **\t║ ANY TASKS ADDED ║**  **\t╚════════════════════╝**  **""";**  **}**  **else {**  **return elements.toString();**  **}**  **}** |
| |  |  | | --- | --- | | **Name:** | getElementsAsArray () | | **Description** | It stores all the elements of the hash table in a single array of nodes. If there is more than one node stored in an index, then it respect that order in the new array | | **Input** | none | | **Return** | (NodesArray)  HashEntry<K,V> allElements | | **public HashEntry<K,V>[] getElementsAsArray2(){**  **HashEntry<K,V>[] allElements = new HashEntry[this.existingNodes];**  **int j = 0;**  **for(int i = 0; i < table.length; i++) {**  **if (table[i] != null) {**  **allElements[j] = new HashEntry<>(table[i].getKey(),table[i].getValue());**  **j++;**  **HashEntry<K,V> current = table[i].getNext();**  **while (current != null) {**  **allElements[j] = new HashEntry<>(current.getKey(),current.getValue());**  **current = current.getNext();**  **j++;**  **}**  **}**  **}**  **return allElements;**  **}** |
| |  |  | | --- | --- | | **Name:** | showArray () | | **Description** | It shows the value of all the elements stored in the array previously generated in getElemenstAsArray(). | | **Input** | none | | **Return** | String: all the values of each stored in the array stored in one single string chain. | | **public String showArray2(){**  **String msg = "";**  **HashEntry<K,V>[] allElements = getElementsAsArray2();**  **if(allElements.length != 0) {**  **for(HashEntry<K,V> element : allElements){**  **if(element != null){**  **msg += "\n\t" + element.getValue().toString() ;**  **}**  **else {**  **msg += "\n\n\tnull";**  **}**  **}**  **}**  **else {**  **msg += "\n\tEMPTY";**  **}**  **msg += "\n\t" + allElements.length;**  **return msg;**  **}** |

|  |  |
| --- | --- |
| **Stack Subroutines** | |
| **Subroutine Specification** | **Construction** |
| |  |  | | --- | --- | | **Name:** | Stack() | | **Description** | It instantiates a new Stack with no elements and size 0 | | **Input** | none | | **Return** | none  (Stack Object instantiated) | | **public Stack() {**  **this.top=null;**  **this.size=0;**  **}** |
| |  |  | | --- | --- | | **Name:** | push() | | **Description** | adds a new element into the stack | | **Input** | T element | | **Return** | none (element added to the stack) | | **public void push(T element) {**  **StackNode<T> created=new StackNode<T>(element);**  **if(top==null) {**  **top = created;**  **}else{**  **top.setTop(created);**  **created.setBottom(top);**  **top=created;**  **}**  **size++;**  **}** |
| |  |  | | --- | --- | | **Name:** | pop() | | **Description** | returns the first element in the structure and deletes it from the data structure | | **Input** | none | | **Return** | output (T): element at the top of the stack  null: if the stack has no elements | | **public T pop() {**  **if(top==null) {**  **return null;**  **}else {**  **T output = top.getContent();**  **StackNode<T> newKing = top.getBottom();**  **if (newKing != null)**  **newKing.setTop(null);**  **top.setBottom(null);**  **top = newKing;**  **size--;**  **return output;**  **}**  **}** |
| |  |  | | --- | --- | | **Name:** | peek() | | **Description** | returns the first element from the stack without deleting it | | **Input** | none | | **Return** | top (T): element at the top of the stack  null: if the stack has no elements | | **public T peek() {**  **return top!=null? top.getContent(): null;**  **}** |
| |  |  | | --- | --- | | **Name:** | isEmpty() | | **Description** | returns whether the stack has elements or not | | **Input** | none | | **Return** | (boolean)  true: the stack has no elements, meaning the top is null  false: the stack has elements, meaning the top is not null. | | **public boolean isEmpty() {**  **return top==null;**  **}** |
| |  |  | | --- | --- | | **Name:** | getSize() | | **Description** | returns the number of elements saved in the data structrue | | **Input** | none | | **Return** | (int)  number of elements saved in the stack | | **public int getSize() {**  **return this.size;**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | getTop() | | **Description** | returns the element saved at the top of the stack | | **Input** | none | | **Return** | (StackNode<T>)  top: element at the top of the stack  null: if the stack is empty | | **public StackNode<T> getTop() {**  **return top;**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | getTop() | | **Description** | returns the element saved at the top of the stack | | **Input** | none | | **Return** | (StackNode<T>)  top: element at the top of the stack  null: if the stack is empty | | **public StackNode<T> getTop() {**  **return top;**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | setTop() | | **Description** | sets a new Node as the top of the stack | | **Input** | StackNode<T> top | | **Return** | none  (new StackNode<T> set as the top of the Stack) | | **public void setTop(StackNode<T> top) {**  **this.top = top;**  **}** |

**MinHeap Subroutines**

|  |  |
| --- | --- |
| **Subroutine Specification** | **Construction** |
| |  |  | | --- | --- | | **Name:** | MinHeap() | | **Description** | It instantiates a new Heap with no elements | | **Input** | none | | **Return** | none  (MinHeap Object instantiated) | | **public MinHeap() {**  **heap = new ArrayList<>();**  **}** |
| |  |  | | --- | --- | | **Name:** | insert() | | **Description** | adds a new element into the MinHeap | | **Input** | T element | | **Return** | none (element added to the heap, keeping its property) | | **public void insert(T element) {**  **heap.add(element);**  **int index = heap.size() - 1;**  **while (index > 0) {**  **int parentIndex = (index - 1) / 2;**  **if (heap.get(index).compareTo(heap.get(parentIndex)) < 0) {**  **// Swap with parent**  **T temp = heap.get(index);**  **heap.set(index, heap.get(parentIndex));**  **heap.set(parentIndex, temp);**  **index = parentIndex;**  **} else {**  **break;**  **}**  **}**  **}** |
| |  |  | | --- | --- | | **Name:** | peekMin() | | **Description** | returns the first element in the structure (the “smallest” one) | | **Input** | none | | **Return** | (T): element at the beginning of the heap  null: if the heap has no elements | | **public T peekMin() {**  **if (heap.isEmpty()) {**  **return null;**  **}**  **return heap.get(0);**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | isEmpty() | | **Description** | returns whether the heap has elements or not | | **Input** | none | | **Return** | (boolean)  true: the stack has no elements, meaning the ArrayList<T> is empty  false: the heap has elements, meaning the ArrayList<T> is not empty | | **public boolean isEmpty() {**  **return heap.isEmpty();**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | addElements() | | **Description** | adds a group of elements to the MinHeap while keeping its property | | **Input** | ArrayList<T> elements | | **Return** | none  (elements added to the heap) | | **public void addElements(ArrayList<T> elements) {**  **for (T element : elements) {**  **insert(element);**  **}**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | extractMin() | | **Description** | returns the smallest element from the Heap and deletes it from the structure | | **Input** | none | | **Return** | (T)  min: smallest element from the heap  null: if the heap has no elements | | **public T extractMin() {**  **if (heap.isEmpty()) {**  **return null;**  **}**  **T min = heap.get(0);**  **heap.set(0, heap.get(heap.size() - 1));**  **heap.remove(heap.size() - 1);**  **int index = 0;**  **while (true) {**  **int leftChildIndex = 2 \* index + 1;**  **int rightChildIndex = 2 \* index + 2;**  **int largest = index;**  **if (leftChildIndex < heap.size() && heap.get(leftChildIndex).compareTo(heap.get(largest)) > 0) {**  **largest = leftChildIndex;**  **}**  **if (rightChildIndex < heap.size() && heap.get(rightChildIndex).compareTo(heap.get(largest)) > 0) {**  **largest = rightChildIndex;**  **}**  **if (largest != index) {**  **// Swap with the largest child**  **T temp = heap.get(index);**  **heap.set(index, heap.get(largest));**  **heap.set(largest, temp);**  **index = largest;**  **} else {**  **break;**  **}**  **}**  **return min;**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | getHeap() | | **Description** | returns all of the elements from the Heap | | **Input** | none | | **Return** | (ArrayList<T>)  heap: list of elements from the heap  null: if the heap is empty | | **public ArrayList<T> getHeap() {**  **return heap;**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | setHeap() | | **Description** | set new elements as part of the heap structure | | **Input** | ArrayList<T> heap | | **Return** | none  (new heap set) | | **public void setHeap(ArrayList<T> heap) {**  **this.heap = heap;**  **}** |

**Queue Subroutines**

|  |  |
| --- | --- |
|  | |
| **Subroutine Specification** | **Construction** |
| |  |  | | --- | --- | | **Name:** | Queue() | | **Description** | It instantiates a new Queue with top and last with the value of null and size 0 | | **Input** | none | | **Return** | none  (Queue Object instantiated) | | **public Queue() {**  **this.top = null;**  **this.last = null;**  **this.size = 0;**  **}** |
| |  |  | | --- | --- | | **Name:** | add() | | **Description** | adds a new element into the Queue | | **Input** | T element | | **Return** | void | | **public void add(T element){**  **QueueNode<T> newNode = new QueueNode<>(element);**  **// The list is empty**  **if(top == null){**  **top = newNode;**  **}**  **else{**  **this.last.setNext(newNode);**  **newNode.setPrevious(this.last);**  **}**  **last = newNode;**  **size++;**  **}** |
| |  |  | | --- | --- | | **Name:** | poll() | | **Description** | returns the first element that was added to the structure and deletes it from the data structure | | **Input** | none | | **Return** | output (T): the first element that was added. (Top)  null: if the Queue has no elements | | **public T poll(){**  **if(top == null){**  **return null;**  **}**  **else {**  **T firstOut = top.getContent();**  **if(top == last){**  **top = null;**  **last = null;**  **}**  **else {**  **QueueNode<T> newTop = top.getNext();**  **newTop.setPrevious(null);**  **top.setNext(null);**  **top = newTop;**  **}**  **size--;**  **return firstOut;**  **}**  **}** |
| |  |  | | --- | --- | | **Name:** | peek() | | **Description** | returns the first element from the Queue without deleting it. (Top) | | **Input** | none | | **Return** | top (T): top element content.  null: if the Queue has no elements | | **public T peek(){**  **return top.getContent();**  **}** |
| |  |  | | --- | --- | | **Name:** | isEmpty() | | **Description** | returns whether the Queue has elements or not | | **Input** | none | | **Return** | (boolean)  true: the Queue has no elements, meaning the size = = 0  false: the Queue has elements, meaning  size != 0 | | **public boolean isEmpty(){**  **return size == 0;**  **}** |
| |  |  | | --- | --- | | **Name:** | getSize() | | **Description** | returns the number of elements saved in the data structure | | **Input** | none | | **Return** | (int)  number of elements saved in the Queue | | **public int getSize() {**  **return size;**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | getTop() | | **Description** | returns the element saved at the top of the stack | | **Input** | none | | **Return** | (QueueNode<T>)  top: element at the top of the Queue  null: if the Queue is empty | | **public QueueNode<T> getTop() {**  **return top;**  **}** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Name:** | showQueue() | | **Description** | returns the toString of all the elements in the Que | | **Input** | none | | **Return** | String with all the toStrings.  if the Queue is empty returns a message stating the queue is empty | | **public String showQueue(){**  **if(this.top == null){**  **return "Queue is empty";**  **}**  **else{**  **return showQueue(top);**  **}**  **}** |