INTEGRATIVE TASK
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

September 20th, 2023

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

ENGINEERING DESIGN METHOD

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

Client **User**	Dr. Marlon Gomez Victoria Users of to-do lists
Functional Requirements	- RF1: Add tasks/reminders - RF2: Modify tasks/reminder - RF3: Delete tasks/reminder - RF4: View tasks/reminder - RF5: Register actions - RF6: Undo actions - RF7: Manage tasks according to their level of importance
Problem Context	*The task management system allows the user to add, organize and manage their missing tasks and reminders. It must also save a registration user's actions and allow them to undo said actions.*
Non-Functional requirements	- The product must include the design of the classes, the data structures and test cases. - The time and algorithmic complexity of some of the methods of the software must be analyzed. - RN3: All tasks and remind must be stored in a hash table - RN4: The User interface has to be easy to use by all kinds of users

Identifier and Name	*RF1: Add tasks and reminders*		
Summary	*The system must allow the user to add a tasks or reminder, by entering the title of the tasks, a description, the due date and type of priority.*		
Input	**Input name**	**Data type**	**Valid condition**
	title	String	*Can't be empty.*
	description	String	*Can't be empty*
	dueDate	Calendar	*Can't be empty. Must have correct date format*
	type	enum	*Can't be empty. Must be either "PRIORITY" or "NOT PRIORITY"*
Result or Postcondition	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.		
Output	**Output name**	**Data type**	**Format**
	message	String	*Confirmation message of whether the tasks were saved or not*

Identifier and Name	*RF2: Modify tasks/reminders*		
Summary	*The system must allow the user to modify a task by changing one of its attributes and entering the key of the task and its new content.*		
Input	**Input name**	**Data type**	**Valid condition**
	key	String	*Must correspond to one of the tasks registered by the user.*
	attribute	int	*Must be an option corresponding with one of the modifiable attributes (1: title, 2: description, 3: date, 4: priority)*
	content	T	*Must correspond to the data type of the attribute to be modified*
Result or Postcondition	After the system receives the data entered by the user, it searches the task with the key entered. If the task is found and it non-priority, the system checks if its the first in line, or if it is priority, it checks that it is the most important. If those cases are presented, 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 or can't be modified. The action is saved in the list of actions.		
Output	**Output name**	**Data type**	**Format**
	message	String	*Confirmation message of whether the tasks was modified or not*

Identifier and Name	*RF3: Delete tasks/reminders*		
Summary	*The system must allow the user to delete a task by entering the key corresponding to the task to be deleted*		
Input	**Input name**	**Data type**	**Valid condition**
	key	int	*Must correspond to one of the tasks registered by the user.*
Result or Postcondition	After the system receives the data entered by the user, it searches for the task with the key entered.If the task is found and it non-priority, the system checks if its the first in line, or if it is priority, it checks that it is the most important. If those cases are presented, 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 or can't be deleted. The action is saved in the list of actions.		
Output	**Output name**	**Data type**	**Format**
	message	String	*Confirmation message of whether the task was deleted or not*

Identifier and Name	*RF4: View tasks/reminders*		
Summary	*The system must allow the user to view their tasks ordered, whether it is by priority or by date.*		
Input	**Input name**	**Data type**	**Valid condition**
	typeOfPriority	int	*Can't be empty. Must correspond to one of the following options:* *1. By Date* *2. By Priority*
Result or Postcondition,	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 its 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.		
Output	**Output name**	**Data type**	**Format**
	tasks	String	*If there are no tasks, "NO TASKS" will be displayed.* *Else, a list of all the tasks will be ordered according to the criteria chosen by the user.*

Identifier and Name	*RF5: Register action*		
Summary	*The system must save the actions done by the user.*		
Input	**Input name**	**Data type**	**Valid condition**
	N/A	N/A	*N/A*
Result or Postcondition	After the user does any action, an id for said action is created and saved in a stack.		
Output	**Output name**	**Data type**	**Format**
	N/A	N/A	*N/A*

Identifier and Name	*RF6: Undo action*		
Summary	*The system must allow the user to undo the last action made by selecting the option.*		
Input	**Input name**	**Data type**	**Valid condition**
	N/A	N/A	*N/A*
Result or Postcondition	After the user chooses said option, the system accesses the stack of actions, pops the first action in the stack and develops the corresponding methods to reverse the action.		
Output	**Output name**	**Data type**	**Format**
	message	String	*Information of the action that was just deleted.*

Identifier and Name	*RF7:* Manage tasks according to their level of importance		
Summary	*The task and reminder have to be shown to the user in two subcategories: priorities and non-priorities. Prioritizing tasks involves storing them in a priority queue, ensuring that the most important tasks are displayed first. Conversely, non-priority tasks are organized in a queue, allowing users to manage them based on their order of arrival. This system efficiently handles task prioritization and management.*		
Input	**Input name**	**Data type**	**Valid condition**
	N/A	N/A	*N/A*
Result or Postcondition	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.		
Output	**Output name**	**Data type**	**Format**
	Priority Tasks List	String	*List of all the existing priority tasks ordered*
	Non-Priority Tasks List	String	*List of all the existing non-priority tasks*

FASE 2: COMPILATION OF NECESSARY INFORMATION

Important Terminology

Hash Table:

A hash table is a data structure known for its effectiveness in the implementation of dictionaries. It uses a hash function (which can be defined according to diverse methods) to compute a *key (that serves as an array's index)*, where the *data* or *value* will be addressed. A hash table typically uses an array of size proportional to the number of keys actually stored. "Chaining" methods are used as a way to handle "collisions," in which more than one key maps to the same slot.

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.

Heap Sort:

Heap sort is a comparison based sorting technique on binary heap data structure. It is similar to selection sort in which we first find the maximum element and put it at the end of the data structure. Then repeat the same process for the remaining items.

Stack

A stack is a linear data structure that stores items in a Last- In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. Here, the only possible actions are push (adding an element at the top of the structure), pop (remove an element from the top of the structure), peek (see the element at the top of the structure) and search (get the position of a certain value in the stack).

Queue:

A queue is a linear data structure that stores items in a First- In/First-Out (FIFO) manner. Here, the new values can be stored at the end of the list and the element to be managed can only be the one at the top of the list. The only possible actions are add (inserts the specified element at the end of the queue), peek (see the element at the top of the structure) and remove (remove an element from the top of the structure).

Priority Task:

Category of tasks/reminders that have a type of priority, and have to be managed according to their level of importance. The most important task must be managed first in order to manage the rest.

Non-Priority Task:

Category of tasks/reminders that don’t have a type of priority assigned and must be managed according to their arrival order (FIFO-First In/First Out).

Actions:

An action serves as the history of the program to be developed. Each one corresponds to a record of the decisions and changes made by the user. Said actions have the possibility of being reversed, which means that the last action made could be undone; in that order, actions are stored in a queue data structure (LIFO-Last In/First Out).

FASE 3: RESEARCH OF CREATIVE SOLUTIONS

Alternative 1

1. Storing Tasks and Reminders: Using hash tables to store tasks and reminders, each of which has a unique identifier that will serve as a key for storing them in the table. A generic Abstract Data Type (ADTs), will be implemented to define both Reminders and Tasks.

- Reminder: Id, Memo, Time, Location, Priority
- Task: Id, Title, Due Date, Description, Location, Priority

2. User Interface: The user interface will be shown in the console, using print strings and scanner methods to show the available options and obtain the response of the user. All available options for each requirement in the system will be organized by menus.

3. Priorities Management:

Two types of categories are defined: priority and non-priority. Priority tasks and reminders will be stored in a priority queue, sorting by heapsort the priority tasks and reminders according to its importance level. Secondly, a non priority queue (FIFO) will be used to store those non-priority reminders or tasks.

4. Undoing Actions Method: Available actions in the software (add, modify or delete a task or reminder). Those actions will be stored in a stack (LIFO), an undo method is going to be defined and will allow the user to undo any type of action; when used, the system will return to its previous state before the user performs the last action.

Alternative 2

1. Storing Tasks and Reminders: ArrayList will be used to store tasks and reminders, each of which has a unique identifier. Both tasks and reminders can be considered as the only type of (ADTs), that will differ only in their type attribute. The data stored in these ADTs are ID, Title, Due Date, Description, Location, and Priority.

2. User Interface: JavaFX is a modern and practical 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.

3. Priorities Management: Two types of categories are defined: priority and non-priority. Properties reminders or tasks will be stored in a binary tree list, whose elements are sorted according to their importance. Secondly, a queue (FIFO) will be used to store those non-priority reminders or tasks.

4. Undoing Actions Method: Actions will be ADTs that will store a complete status of the whole system, which will be stored in a stack (LIFO), an undo method is going to be defined and will allow the user to undo any type of action, and restore the system to its previous status.

Alternative 3

1. Storing Tasks and Reminders: An Open Addressing table is going to be used to store both tasks and reminders. Both Reminder and Task will be defined by their own class with the same name, and attributes of primitive data:

- Reminder: Id, Memo, Time, Location, Priority
- Task: Id, Title, Due Date, Description, Location, Priority

2.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.

3. Priorities Management: Two types of categories are defined: priority and non-priority. Properties reminders or tasks will be stored in an arrayList, whose elements are sorted according to their importance; which is defined by the due date attribute. Secondly, a queue (FIFO) will be used to store those non-priority reminders or tasks.

4. Undoing Actions Method: The opposite action to a previous user action will be applied, in the case the user wants to undo something he has done. For example, if the user has deleted a task or reminder, then the undo method is going to apply the register method with the element that has been deleted.

FASE 4: TRANSITION FROM IDEA FORMULATION TO PRELIMINARY DESIGNS

Alternativa 1 (Chosen):

1. Storing Tasks and Reminders:

- When using hash tables of direct addressing, reduce the algorithmic time when adding, searching and deleting an element.
- Using Hash Tables allow us to modify the hash function according to the needs of space demanded by the system
- When using ADTs, attributes of both task and reminders can be adapted to the future needs of the user
- An ADTs allow us to easily manipulate the data referred to tasks and reminders, and adequate them to future changes

2.User Interface:

- Print methods alone lack the necessary features to create interactive and user-friendly interfaces. Displaying menus in the console simplifies the design process, resulting in practical, straightforward, and easily understandable interfaces for system users.
- Console-based menu displays enable organizations to interconnect methods in a practical and efficient manner, facilitating a chain of responsibility among employees.
- Learning to create console menus typically requires less time and effort compared to other design alternatives like JavaFX or Java GUI. Embracing this approach can lead to more efficient and accessible user interfaces.

3.Priorities Management:

- A priority queue ensures efficient task management by addressing high-priority tasks first. When implemented with a heap-based structure like a binary heap, it offers rapid task insertion and removal, optimizing resource allocation.
- When applying a queue (FIFO) to store non-priority tasks or reminders, then it will be easy for the user to consult those activities by its arrival order.

4. Undoing Action Method:

- Defining a new ADTs for user actions will allow the system to store predefined information, susceptible to change, registration, modification and deletion
- Using a stack (LIFO) to store those ADTs allows the system to have an accessible and simple reference to the previous state of a user action

Alternative 2 (Discarded):

1. Storing Tasks and Reminders:

- When using ArrayList to store the ADTs is going to increment the search complicity from O(1) to O(n)
- Using Hash Tables allow us to modify the hash function accordingly to the needs of space demanded by the system
- When suing ADTs, attributed of both task and reminders can be adapted to future needs of the user
- An ADTs allow us to easyfully manipulate the data referred to tasks and reminders, and adequate them to future changes

2.User Interface:

- Modern and Rich UI: JavaFX, offers modern and visually appealing user interfaces compared to Swing, giving the programmer more options and animations.
- JavaFX allows the application of CSS styling to the UI components, making it effortless to achieve a consistent and visually appealing look and feel across the program.
- A rich set of UI controls such as buttons, text fields, tables, and charts are offered and highly customizable. Allowing the developer to create a different and unique design of his application or program
- Multimedia Integration: JavaFX can interact with web technologies, audio, media-rich applications, educational software and entertainment applications.
- While Java fx is a good design tool, it requires some prior knowledge of its operation and structure. This will take some time for the developers of a programme to learn.

3. Priorities Management

- Not having s subcategory in the level of importance of priority tasks, makes it almost impossible to sort in a binary tree tasks according to their significance
- As tasks and reminders do not have a hierarchical dependence, and are going to be sorted according to a level of importance and not a numerical key value, binary trees will not be an adequate data structure to store these elements; as tasks and reminders can be stock up in a linear database form.

4. Undoing Action Method:

- Defining a new ADTs for user actions will allow the system to store predefined information, susceptible to change, registration, modification and deletion
- By using a stack (LIFO) to store those ADTs allow the system to have an accessible and simple reference to the previous state of user action
- An unnecessary use of memory is evidenced as this ADTs keep the information of the system as a whole, when just remembering the user action is requested

Alternative 3 (Chosen):

1. Storing Tasks and Reminders:

- When using an Open Addressing Table to store the ADTs will increment the algorithm complexity from O(1) to O(n)
- Creating just one ADTs for both reminders and tasks will note allow future specific changes (new methods, new attributes, etc) that could be applicable to tasks but not to reminders, and vice versa.

2. User Interface:

- Easy of Learning: The Gui package allows the developing team to create a User-Friendly Interaction more practically, as it can instantiate clickable buttons, fill-in forms, and graphical elements. Also, it is a facile Java tool for occasional programmers who are not familiar with it
- Improved Accessibility: The software interface can be adapted to the user context and abilities, presenting a clear data visualization and reducing errors.
- Customization: GUIs can be customized to each system, application or program content and context.

3. Priorities Management

- By creating an ArrayList of priority activities, a sorted method can be easily implemented to organize elements according to their importance
- When applying a queue (FIFO) to store non-priority tasks or reminders, then it will be easy for the user to consult those activities by its arrival order.
- Sorting priorities, according to the due date will not organize tasks and reminders suitably to their importance and relevance, but to the most urgent activity that has to be completed in terms of time

4. Undoing Action Method:

- Applying the opposite method to the previous user action, will demand storing specific information for the various cases (actions) that the user performs
- The modifying method will not have a direct opposite action to compare, thus it will demand creating a special method that stores the previous state of the task or reminder

PHASE 5: EVALUATION AND SELECTION OF THE SOLUTION

5.1 Criteria Evaluation Definition in terms of Quality:

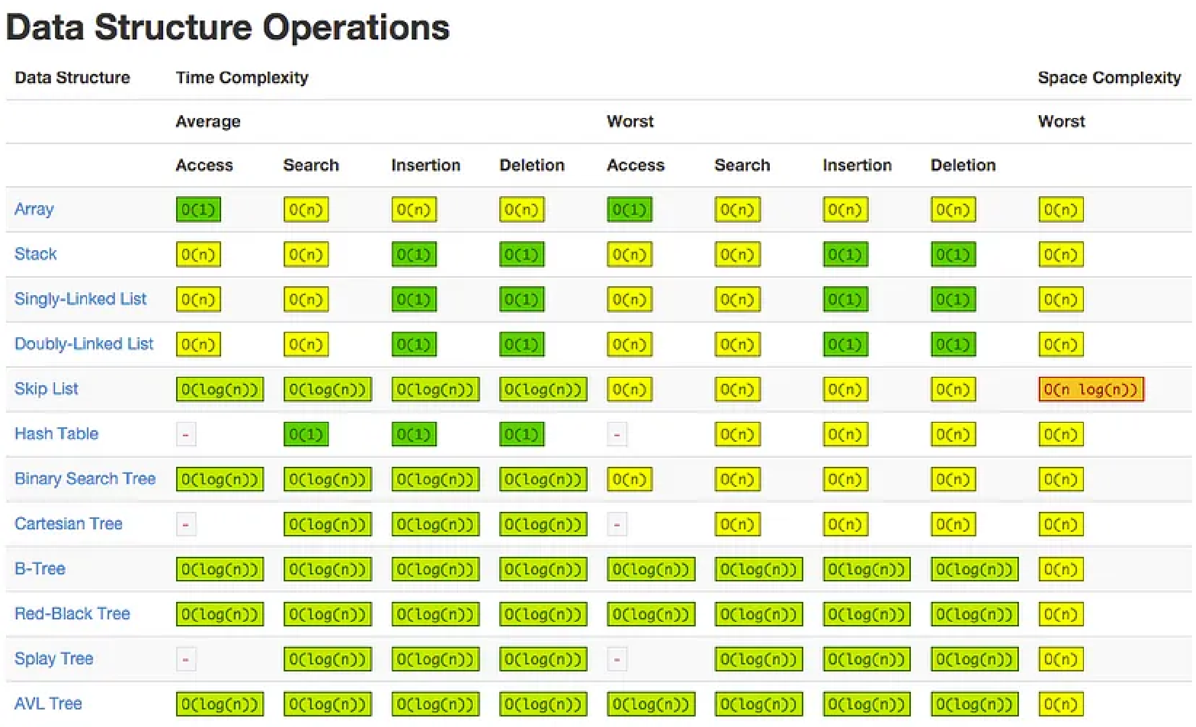
CRITERIA QUALITY EVALUATION DEFINITION		
Criteria	**Definition**	**Evaluation Scale**
Functionality	Assess how well the software meets the required functionality, including precision, adequacy, interoperability, conformance, and security.	-1: Does not meet functionality requirements -2: Partially meets 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. Storing Tasks and Reminders	4	4	5	5
2. User Interface	4	4	4	4
3. Priorities Management	4	4	4	4
4. Undoing Actions Method	5	3	5	4
Total Punctuation	**17**	**15**	**18**	**17**

ALTERNATIVE 3 EVALUATION				
Requisit	**Functionality Punctuation**	**Reliability Punctuation**	**Usability Punctuation**	**Efficiency Punctuation**
1. Storing Tasks and Reminders	4	3	4	3
2. User Interface	4	3	3	4
3. Priorities Management	3	4	3	4
4. Undoing Actions Method	2	2	3	2
Total Punctuation	**13**	**12**	**13**	**13**

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



Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Font:Medium
https://medium.com/@Hollyzhou/data-structure-the-big-o-notation-e3e2405bb8eb

BIG (O) EVALUATION SCALE		
Complexity	**Definition**	**Punctuation**
Quadratic	$O(n^2)$	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**
Storing Tasks and Reminders		
Add Tasks or Reminders Efficiency	This criterion evaluates how efficiently the system allows users to add new tasks or reminders. It assesses the speed and resource utilization when inserting new items into the storage structure, such as hash tables, while maintaining data integrity.	BIG O CRITERIA
Accessing Tasks or Reminders Efficiency	This criterion measures the system's efficiency in retrieving and accessing tasks or reminders. It assesses the speed and resource usage when querying and displaying stored items, ensuring that users can quickly and effectively find the information they need.	BIG O CRITERIA
Deleting Tasks or Reminders Efficiency	This criterion examines how efficiently the system handles the removal of tasks or reminders. It assesses the speed and resource utilization when deleting items from the storage structure, ensuring that the process is swift and doesn't compromise the system's performance.	BIG O CRITERIA
Priorities Management		
Priority Tasks and Reminders Management Efficiency	This criterion focuses on the efficiency of managing priority tasks and reminders. It evaluates how well the system organizes and retrieves high-priority items, ensuring	BIG O CRITERIA

	they are readily available for the user when needed, and whether this management incurs minimal overhead.	
Non-Priority Tasks and Reminders Management Efficiency	This criterion assesses the system's efficiency in managing non-priority tasks and reminders. It evaluates how well the system handles the organization and retrieval of less critical items, utilizing data structures like queues to maintain efficiency and user satisfaction.	BIG O CRITERIA
Undoing Action Methods		
Undoing Action Method Efficiency	This criterion evaluates the efficiency of the system's "undo" functionality. It assesses how quickly and reliably the system can revert to a previous state after a user-initiated action, ensuring a seamless user experience and minimal disruption to the workflow.	BIG O CRITERIA

Criteria	Functionality	Reliability	Usability	Efficiency	**Total**
Alternative 1	**17**	**15**	**18**	**17**	**67**
Alternative 3	**13**	**12**	**13**	**13**	**51**

5.2.1 Evaluation of Structures in Chosen Alternatives in terms of Algorithm Complexity (Worst Case):

Alternative 1 Evaluation

ALTERNATIVE 1 (Storing Tasks and Reminders) HASH TABLE STRUCTURE: Collisions Managed with Double Linked List	
Criteria	**Evaluation Scale (Big-O)**
Add Tasks or Reminders Efficiency	2
Accessing Tasks or Reminders Efficiency	2
Deleting Tasks or Reminders Efficiency	2
Total Punctuation	**6**

ALTERNATIVE 1 (Priorities Management) Priority Queue (Heap Sort)	
Criteria	**Evaluation Scale (Big-O)**
Priority Tasks and Reminders Management Efficiency	3
Total Punctuation	**3**

ALTERNATIVE 1 (Priorities Management) Non-Priority Queue	
Criteria	**Evaluation Scale (Big-O)**
Non-Priority Tasks and Reminders Management Efficiency	2
Total Punctuation	**2**

ALTERNATIVE 1 (Priorities Management) Stack Structure (LIFO)	
Criteria	**Evaluation Scale (Big-O)**
Undoing Action Method Efficiency	2
Total Punctuation	**2**

Alternative 3 Evaluation

ALTERNATIVE 3 (Storing Tasks and Reminders) Hash Table: Opening A Dressing Table	
Criteria	**Evaluation Scale (Big-O)**
Add Tasks or Reminders Efficiency	2
Accessing Tasks or Reminders Efficiency	2
Deleting Tasks or Reminders Efficiency	2
Total Punctuation	**6**

ALTERNATIVE 3 (Priorities Management) ArrayList	
Criteria	**Evaluation Scale (Big-O)**
Priority Tasks and Reminders Management Efficiency	2
Total Punctuation	**2**

ALTERNATIVE 3 (Priorities Management) **Non-Priority Queue**	
Criteria	**Evaluation Scale** **(Big-O)**
Non-Priority Tasks *and Reminders Management Efficiency*	2
Total Punctuation	**2**

Comparison of Alternative 1 and Alternative 3 Algorithmical Time Complexity:

Criteria	**Storing Tasks and Reminders**			**Priorities Management**		**Undoing Action Method**	**Total**
	Add Tasks or Reminders Efficiency	*Accessing Tasks or Reminders Efficiency*	*Deleting Tasks or Reminders Efficiency*	*Priority Tasks and Reminders Management Efficiency*	*Non-Priority Tasks and Reminders Management Efficiency*	*Undoing Action Method Efficiency*	
Alternative 1	**2**	**2**	**2**	**3**	**2**	**2**	**13**
Alternative 3	**2**	**2**	**2**	**2**	**2**	**-**	**10**

5.3 Selection of the best Alternative:

Alternative 1:

This solution has been selected due to its comprehensive approach to task and reminder management. It leverages hash tables for efficient storage, implements a user-friendly interface by console for broad platform compatibility, and introduces a sophisticated priority management system with both priority and non-priority categories. Additionally, the inclusion of an "Undo" feature through a stack ensures user flexibility and error correction. Overall, this solution excels in functionality, usability, and practicality, earning it the highest score.

PHASE 6: PREPARATION OF REPORTS AND SPECIFICATIONS

6.1 General Problem Specification:

Design a task and reminder management system allowing users to add, organize, and manage their pending tasks and reminders. The system comprises components such as task and reminder storage using a hash table with unique identifiers as keys, a user-friendly interface for adding, modifying, and deleting tasks and reminders, and sorting options based on deadlines or priorities utilizing heapsort. Tasks are categorized into "Priority" and "Non-priority," where prioritized tasks are managed through a priority queue, ensuring important tasks take precedence. Non-priority tasks are organized on a first-in, first-out (FIFO) basis. Additionally, implement an "Undo" feature using a stack (LIFO) to track user actions, allowing users to revert to the last performed action. This system efficiently addresses task and reminder management needs.

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

Diagram:

Task manager.pdf

Hash Table Task to Implement:

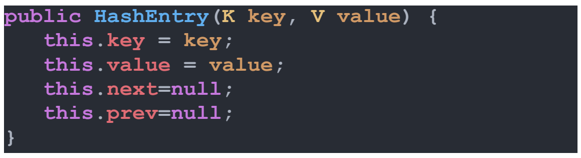
HashEntry

HashEntry Subroutines

Subroutine Specification

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)

Construction



```
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

```
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;
}
```

	null: if the node is not stored in the hash table

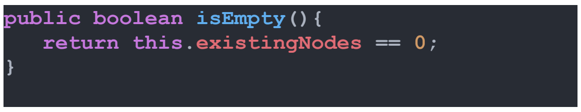
Name:	findValue ()
Description	returns a the value of a node stored in an index that has more than one more stored; managing collisions.
Input	<K> key
Return	<V> value: if the node if founded null: if the node is not stored in the hash table

```
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

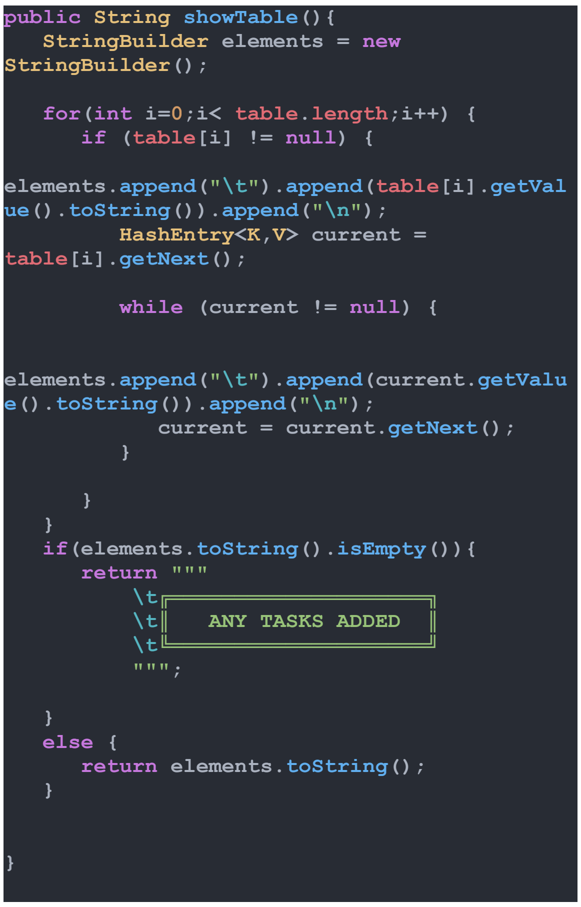
```
public void delete(K key, V value) {
    int index = hashFunction(key);
    if(table[index]==null){
        System.out.println("Node not found!");

    }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].getVal
ue().toString()).append("\n");
         HashEntry<K,V> current =
table[i].getNext();

         while (current != null) {


elements.append("\t").append(current.getValu
e().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].getVa
lue());
         j++;
         HashEntry<K,V> current =
table[i].getNext();
         while (current != null) {

            allElements[j] = new
HashEntry<>(current.getKey(),current.getValu
e());
            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.

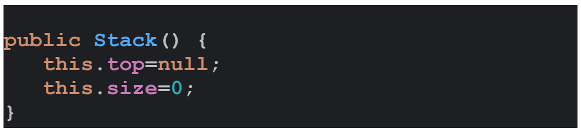
```
public String showArray2(){
   String msg = "";

   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

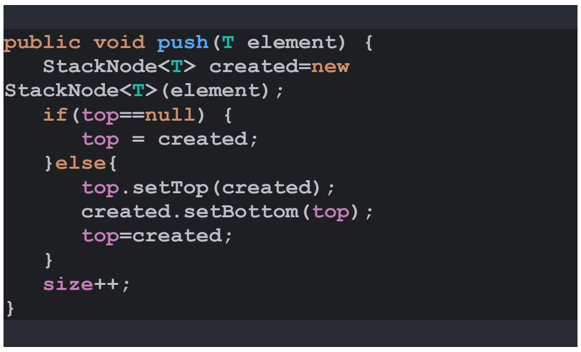
Name:	Stack()
Description	It instantiates a new Stack with no elements and size 0
Input	none
Return	none (Stack Object instantiated)

Construction



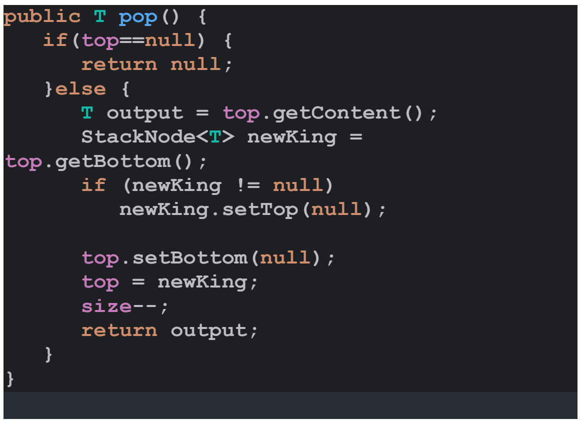
```
public Stack() {
   this.top=null;
   this.size=0;
}
```

Name:	push()
Description	adds a new element into the stack
Input	T element
Return	none (element added to the stack)



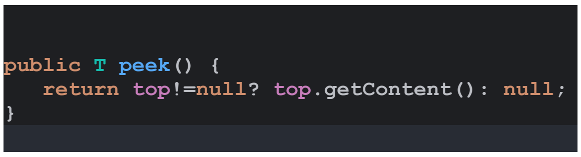
```
public void push(T element) {
   StackNode<T> created=new
StackNode<T>(element);
   if(top==null) {
      top = created;
   }else{
      top.setTop(created);
      created.setBottom(top);
      top=created;
   }
   size++;
}
```

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



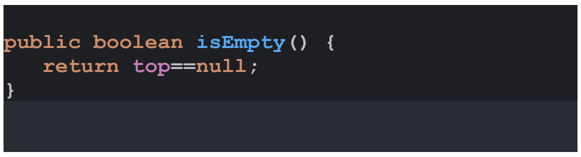
```
public T pop() {
   if(top==null) {
      return null;
   }else {
      T output = top.getContent();
      StackNode<T> newKing =
top.getBottom();
      if (newKing != null)
         newKing.setTop(null);

      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



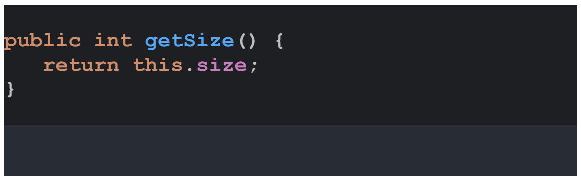
```java
public T peek() {
    return top!=null? top.getContent(): null;
}
```

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.



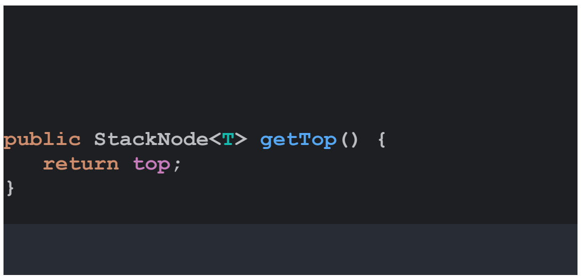
```java
public boolean isEmpty() {
    return top==null;
}
```

Name:	getSize()
Description	returns the number of elements saved in the data structrue
Input	none
Return	(int) number of elements saved in the stack



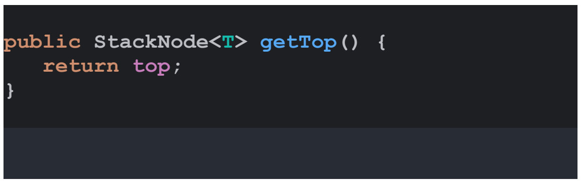
```java
public int getSize() {
    return this.size;
}
```

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



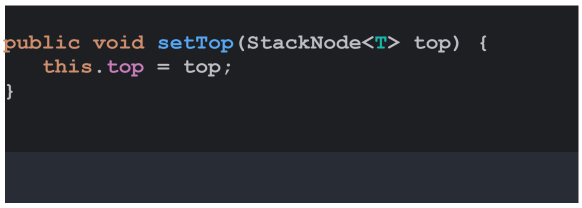
```
public StackNode<T> getTop() {
    return top;
}
```

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)

```java
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)

```java
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(parentIn
dex)) < 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

```java
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

```java
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)

```java
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

```java
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

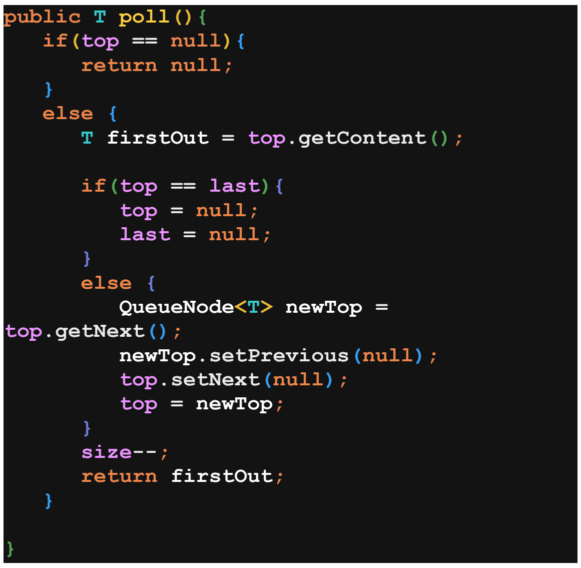
Name:	Queue()
Description	It instantiates a new Queue with top and last with the value of null and size 0
Input	none
Return	none (Queue Object instantiated)

```
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

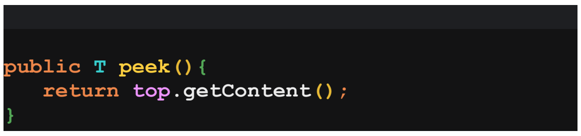
```
public void add(T element){
    QueueNode<T> newNode = new
QueueNode<>(element);
    // The list is empty
    if(top == null){
        top = newNode;
        }
    else{
        this.last.setNext(newNode);
        newNode.setPrevious(this.last);
        }

        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



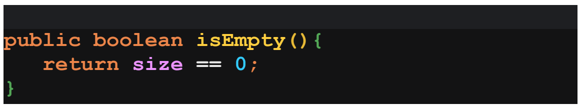
```
public T poll(){
   if(top == null){
      return null;
   }
   else {
      T firstOut = top.getContent();

      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



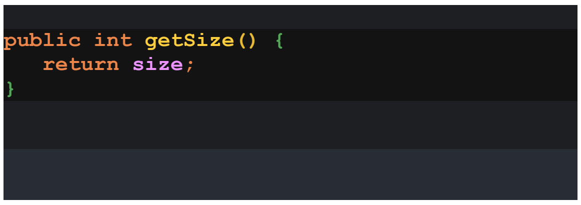
```
public T peek(){
   return top.getContent();
}
```

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



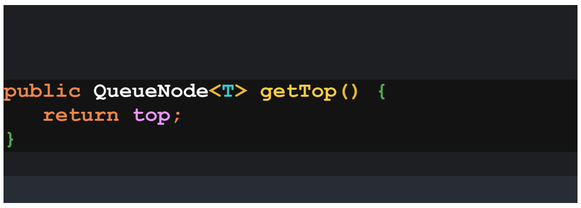
```
public boolean isEmpty(){
   return size == 0;
}
```

Name:	getSize()
Description	returns the number of elements saved in the data structure
Input	none
Return	(int) number of elements saved in the Queue



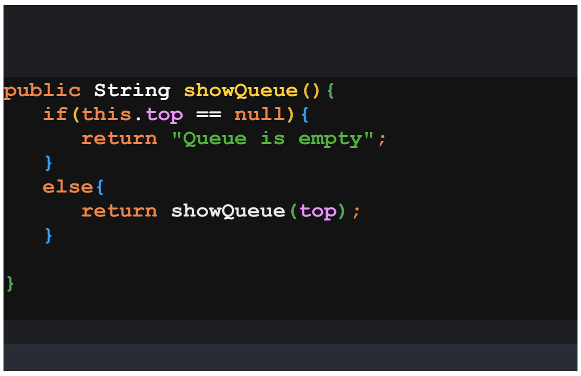
```
public int getSize() {
   return size;
}
```

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);
    }

}
```