**Step 1. Problem Identification**

***Identification of needs and symptoms***

- Users need a task management and reminder system.

- The system should allow users to add, modify, and delete tasks and reminders.

- The system should allow users to enter into the system the information of their tasks and reminders: title, description, deadline, priority, etc.

- Users should have the ability to see a list of all tasks and reminders, sorted by due date or priority

- The system should allow users to manage their tasks according to an order of priority. The most important tasks must be taken care of first.

- Users can also add tasks without a defined priority, which must be addressed on a first-come, first-served basis.

- The system should allow users to undo actions they have taken within the system, giving them the ability to revert changes.

**Unambiguously concrete identification and definition of the problem**

Users require a task and reminder management system that allows them to add, modify, and delete tasks and reminders.

**Functional Requirements**

**SOFTWARE ENGINEERING PROBLEM SPECIFICATION TABLE, identifying the following items**

\*In this case, it was not specified which would be the company or person who ordered the program or the target audience that will make use of the system's functionalities. But we could say for example that the client is the university and the users will be the students, so it would be a system designed by the university to help students manage their time better.

|  |  |
| --- | --- |
| CUSTOMER | \* |
| USER | \* |
| FUNCTIONAL REQUIREMENTS | RF1: Add Task  RF2: Modify Task  RF3: Delete Task  RF4: Add a Reminder  RF5: Modify a reminder  RF6: Delete a reminder  RF7: Organize tasks according to priority  RF8: Undo actions. |
| CONTEXT OF THE PROBLEM | Users require a task and reminder management system that allows them to add, modify, and delete tasks and reminders. |
| NON-FUNCTIONAL REQUIREMENTS |  |

**Functional Requirements Analysis Table (Note: One table for each functional requirement)**

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF1: Add Task | | |
| Summary | The user enters the information of a new task that they are going to add to their management system. You receive the title of the task, its description, the deadline and, if applicable, its priority. As a group we have established that the priority to be handled will be from 1 to 4. With 1 being the least priority and 4 being the most priority. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| Title | String | Must be at least 3 characters long |
| Description | String | Must be at least 3 characters long |
| Deadline | Int | It must have 10 characters. It cannot be less than the current date. |
| priority | Boolean | You must select whether it is a priority or not |
|  | Priority Value | Int | Value between 1 and 4 |
| General activities required to achieve the results | 1. Enter Title 2. Login Description 3. Enter Deadline 4. Indicates if it's a priority 5. Indicates priority level, if applicable. | | |
| Outcome or post-condition | All recorded data is saved and a new task is created in the system. | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| confirmationTask Creation. | Boolean | Enter all data according to variable types |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF2: Modify Task | | |
| Summary | The user modifies the information in a task. To do this, select a task from the task list, then indicate which task information you want to change, and enter the data with which you want to replace the previous information. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| selection | Int | The value entered must be in the range of possible options. |
| Selection of Information to be modified | Int | The value entered must be in the range of possible options. |
| Value of the information to be replaced | String |  |
| General activities required to achieve the results | 1. The user selects a task from the task list. 2. Indicate what you want to change. 3. Enter your information. 4. The change is saved. | | |
| Outcome or post-condition | Save all logged changes. | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| confirmationChangemade. | Boolean |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF3: Delete a task | | |
| Summary | In order for the user to delete a task, the system shows the list of tasks, the user must indicate the task to be deleted according to the list that is shown. Once this is done, the system removes the task from the task list. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| selection | Int | The value entered must be in the range of possible options. |
| General activities required to achieve the results | 1. Enter Task Selection 2. Confirm deletion 3. Delete Task | | |
| Outcome or post-condition | The task is removed from the task list. | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| confirmationEliminationTareal. | Boolean | Enter all data according to variable types |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF4: Add a Reminder | | |
| Summary | The user enters the information of a new reminder that they are going to add to their management system. The title, description, and deadline are received. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| Title | String | Must be at least 3 characters long |
| Description | String | Must have 3 characteristics minimum |
| Deadline | Int | It must have 10 characters. It cannot be less than the current date. |
| General activities required to achieve the results | 1. Enter Title 2. Login Description 3. Enter Deadline | | |
| Outcome or post-condition | All recorded data is saved and a new reminder is created in the system. | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| confirmationAddReminder. | Boolean | Enter all data according to variable types |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF6: Modify Reminder | | |
| Summary | The user modifies the information in a reminder. To do this, select one from the list, then indicate what information you want to change and enter the data with which you want to replace the previous information. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| selection | Int | The value entered must be in the range of possible options. |
| Selection of Information to be modified | Int | The value entered must be in the range of possible options. |
| Value of the information to be replaced | String |  |
| General activities required to achieve the results | 1. The user selects a reminder. 2. Indicate what you want to change. 3. Enter your information. 4. The change is saved. | | |
| Outcome or post-condition | Save all logged changes. | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
| confirmationModifyReminder. | Boolean | Enter all data according to variable types |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF6: Delete a reminder | | |
| Summary | In order for the user to delete a reminder, the system shows the list, the user must indicate the reminder to be deleted. Once this is done, the system removes the reminder from the list. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| selection | Int | The value entered must be in the range of possible options. |
| General activities required to achieve the results | 1.Enter Selection  2.Confirm Deletion  3.Delete Task | | |
| Outcome or post-condition | The reminder is removed from the list. | | |
| Outputs | Ticket name | Type of data | Selection or repetition condition |
| confirmationDeleterReminder. | Boolean | Enter all data according to variable types |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF7: Organize tasks according to priority | | |
| Summary | When the list of tasks and reminders is to be displayed on the screen, the tasks should be ordered with respect to their priority, showing the most important tasks first. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
|  | Print list | Int | The value entered must be in the range of possible options. |
| General activities required to achieve the results | 1. The user indicates that they want to see their to-do list and reminders. | | |
| Outcome or post-condition | The list of tasks and reminders is displayed on the screen. | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
|  | ConfirmationOrganizeTasks | Boolean | Enter the data with the above conditions |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | RF8: Undo Actions | | |
| Summary | The system must have the ability to undo actions. Just like in a Word document, the user should be able to revert to the previous state of the system until it is returned to startup. Therefore, the actions that the user has taken should be reverted to the point that the user indicates. To do this, the user simply selects the *undo* option of the program. The program rolls back an action each time the user selects the option. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
|  | selection | Int | The value entered must be in the range of possible options. (in this case, the undo option) |
| General activities required to achieve the results | 1. The user indicates that they want to undo actions. | | |
| Outcome or post-condition | The program is returned to a previous state, that is, an action taken is rolled back. If the user had added a task, it is deleted, if they had deleted a task, it is re-entered, and so on with each case. | | |
| Outputs | Departure Name | Type of data | Selection or repetition condition |
|  | ConfirmationUndoActions | Boolean |  |

Step 2. Collection of Information

Sources:

<https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/las-mejores-aplicaciones-de-gestion-de-proyectos/>

(IONOS is a software company with more than 30 years of experience in the industry)

<https://www.xataka.com/basics/14-mejores-aplicaciones-para-administrar-tus-tareas-tus-proyectos>

<https://www.ionos.es/startupguide/productividad/gestion-del-tiempo/>

<https://www.freecodecamp.org/news/hash-tables/>

<https://es.wikipedia.org/wiki/Lluvia_de_ideas>

To gather information, we must look for the most relevant concepts for our problem. In this case, let's start by looking at the most used systems or applications for time and task management both in everyday life and for work projects. The top 3 we found were:

1. **Todoist**: One of its most important features is that it's available for almost all types of devices: mobile devices, computers, and smartwatches. As for its functionality, you can create tasks with descriptions and subtasks, choosing the day and time at which they should be completed. It has a tag system to catalog these tasks, and a four-level priority system to decide which of them are most urgent. You can also work as a team and create tasks that repeat over time.
2. **Trello:** Along with Asana, Trello is one of the most popular services for managing tasks when working in a team or at a business level, but it can also be used for personal projects. It stands out above all for its interface, with a structure of boards and cards with which you can manage tasks in a much more visual way. Each board is a project, and each card within a board represents a task to be accomplished. Within each card, you can add descriptions, subtasks, or attach files.
3. **Microsoft To Do:** After buying Wunderlist, one of the best apps for task management, Microsoft designed this task manager. It has a clean and simple interface, being able to create several tasks in it, set deadlines and times for their fulfillment. One of its best features is a main page called *My Day*, which shows everything a person has to do on the present day.

It is not only important to look for task management applications, but to understand the context and why the task we must carry out is important, to understand the context and have a better clarity in the business logic, we investigate about time management and its importance:

Time management is about using your available time as efficiently as possible. Mastering time management has several advantages:

* Increased productivity
* Stress Reduction
* More time for self and family
* Improved work-life balance

These advantages are, at the end of the day, what we hope to achieve with the program we want to design.

For the resolution of the problem posed, it is also necessary to be clear about the structures that we are going to implement for the development of the solution, we will give a brief but detailed information of the most important structures that we are going to use for the solution of the problem:

# Hash Table

A hash table, also known as a hash map, is a data structure that maps keys to values. It works on the concept of hashing, where each key is translated by a hash function into a distinct index in an array. The index functions as a storage place for the corresponding value.

Key points to keep in mind:

Hash function: A function that translates keys to indexes in an array is known as a hash function. The keys must be evenly distributed throughout the array by a good hash function to reduce collisions and ensure fast search speeds.

Load Factor: The load factor of a hash table is determined by how many items are stored there relative to how large the table is. If the load factor is high, the table can be cluttered and have longer search times and collisions.

Collision Resolution: Collisions occur when two or more keys point to the same fix index. There are several techniques for resolving collisions, such as chaining and open steering.

Priority Queue

A priority queue is a type of data structure, similar to a regular queue or stack. Each item in a priority queue has a priority associated with it. Items with high priority are taken care of before items with low priority.

In a priority queue, each item has a priority value associated with it. When an item is added to the queue, it is inserted into a position based on its priority value. For example, if an item with a high priority value is added to a priority queue, it is inserted near the front of the queue, while an item with a low priority value is inserted near the end.

Typical operations supported by a priority queue are:

Insertion: When a new element is inserted into a priority queue, it is evaluated from top to bottom and left to right (abstracting from the array to a tree). However, if the element is not in the right place, then it is compared to the parent node. If the item is not in the correct order, the items are swapped.

Delete: The item that has the highest priority will be deleted first. Therefore, the root node is removed from the queue. It can also be implemented so that the item with the lowest priority is deleted first. It all depends on the implementation and the goal you are looking for. In our case, we want the tasks with the highest priority to be taken care of first.

Priority queues are often used in real-time systems, where the order in which items are processed can have significant consequences.

# Stack

It is a linear data structure that follows the LIFO (Last In, First Out) principle, which means that the last element to enter is the first to leave.

Stacks basically allow two operations: push and pop. When you stack an item, it is added to the top of the stack. When you unstack, the item at the top of the stack is removed.

Stacks are used in the handling of function calls, where each time a function is called it is stacked on the stack and when it finishes the function is unstacked. They are also used to reverse an operation (e.g., the undo function in a text editor) and in the evaluation of mathematical expressions in reverse Polish notation.

Stacks are simple and easy to implement. Insertion and removal operations are quick and efficient. In addition, stacks are useful for solving problems that require an exhaustive search and tracking of a path or route, such as in backtracking.

On a stack, only the item at the top of the stack can be accessed, which can be a disadvantage if intermediate items need to be accessed. In addition, stacks are not designed for efficient search for specific items.

A stack is an essential data structure for programming and is used in a variety of problems and applications.

# Queue

It is a linear data structure that follows the FIFO (First In, First Out) principle, which means that the first element to enter is the first to leave.

The basic operations that can be performed on a queue are: enqueue and dequeue. Gluing an item adds it to the end of the queue, and degluing removes the item at the front of the queue.

The glue is used in various applications such as:

* In operating systems, the queue is used to store processes waiting to be executed.
* In computer networks, queues are used to store packets waiting to be transmitted.
* In printing systems, queues are used to store documents waiting to be printed.

The advantages of queues include their specific processing order, their scalability, and their usefulness in event simulation and planning.

However, they also have some disadvantages. For example, only the first item in the queue can be accessed, which can be a limitation if intermediate items need to be accessed. In addition, the tails are not designed for the efficient removal of intermediate elements.

**Step 3. Search for Creative Solutions**

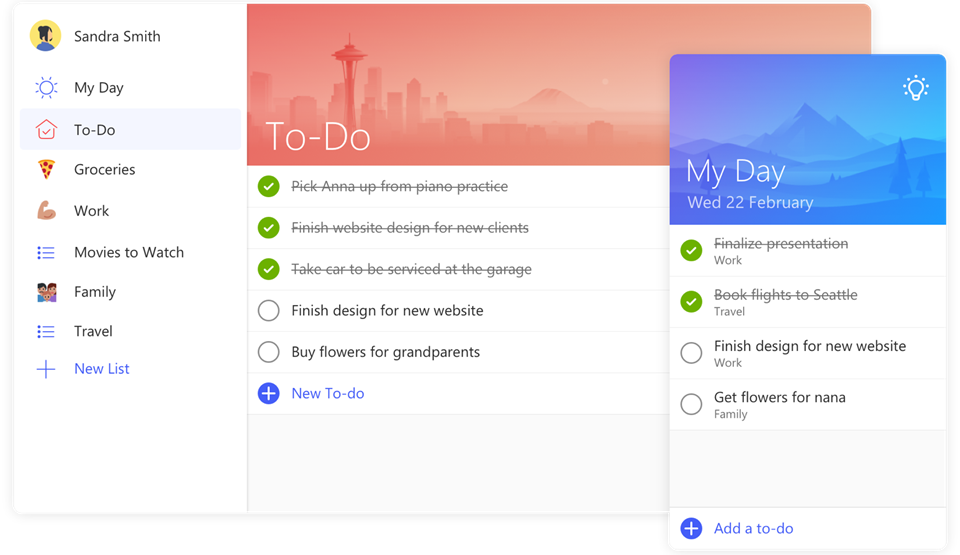
To find creative solutions, we use brainstorming, which is a group work tool that facilitates the emergence of new ideas on a given topic or problem. Brainstorming is a group technique for generating original ideas in a creative environment.

Alternative 1. Physical agenda and google calendar



The first thing we came up with as a group was to make a physical agenda for task management. The student can keep a record of his/her assignments in one of his/her notebooks, assigning priority tasks a criterion defined by him/her. For reminders, you can use the google calendar app or any other similar app.

Alternative 2. Use of any of the above applications



In the information gathering phase, we mentioned some existing apps that help manage time and tasks. The student or user can make use of any of these tools, it will surely be of help because these applications are known for their help in task management.

Alternative 3. Development of a web application

In the compilation of information we realized that one of the features that people appreciate most about an application that helps them in their task management, is to be able to access said application from a wide variety of devices and from anywhere, this is why we thought about the development of a web application; A person could access it from their phone or computer and could even access it from different places such as an office, a library, among others.

Alternative 4. Development of a simple system in Java

Development of a system that meets the stated requirements, using the knowledge we have acquired at the university.

**Step 4. Transitioning from Ideas to Preliminary Designs**

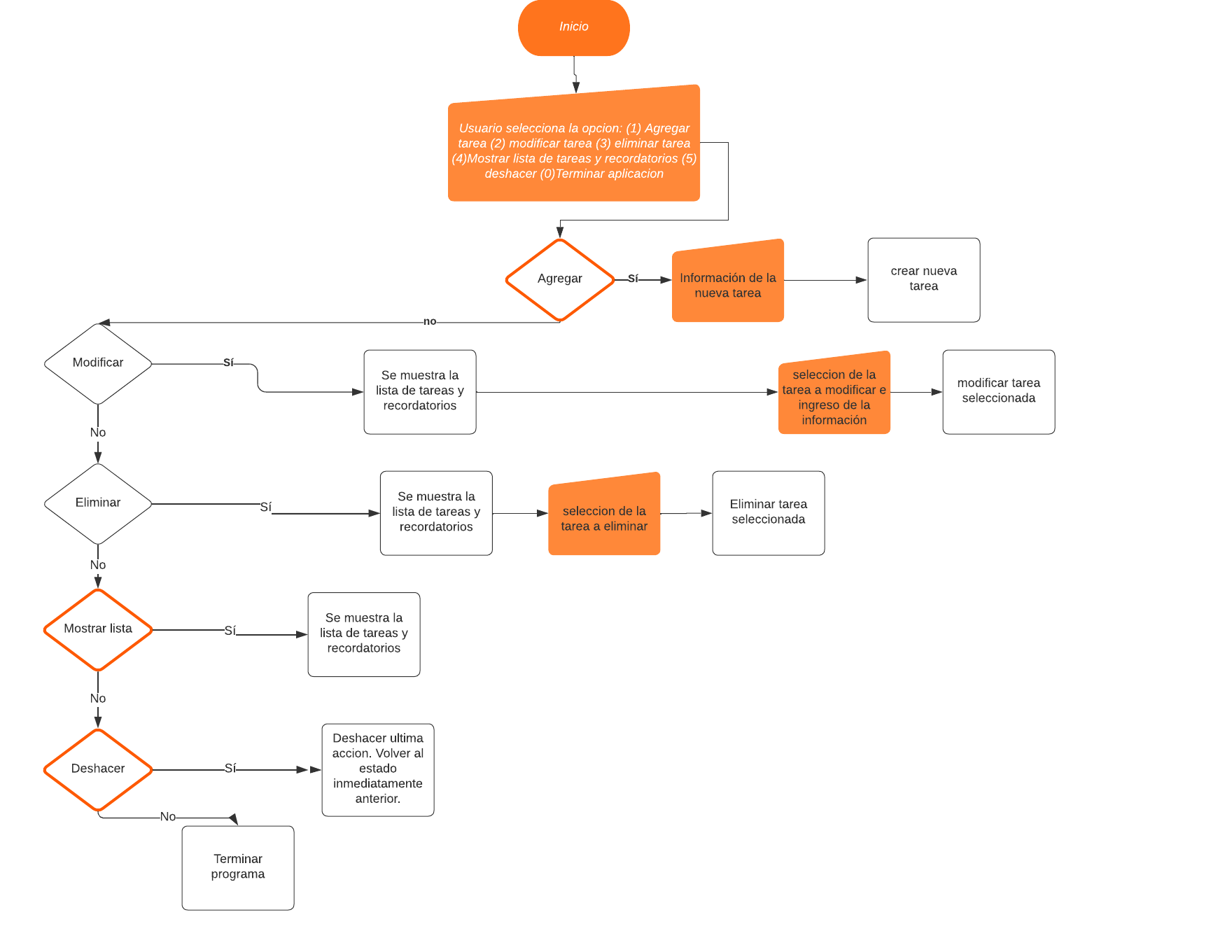
We discard alternatives 1 and 2, because despite being practical, simple and powerful, they do not really require any effort on the part of us students, and since the objective of this work is to test our knowledge, it would not be possible to use them as a solution in this case. We also discarded alternative 3 since we do not have the necessary knowledge to develop a web application with everything it requires: front-end design and logic, data management from the back-end, deployment, among many others. Therefore, let's carefully review Alt 4:

Alternative 4. Development of a simple system in Java

* To develop a program in Java we need to know what data structures we are going to implement in order to be able to solve what the statement asks us to do. That is why we are going to use the structures we investigate in the information gathering phase, as these guarantee an efficient use of resources.
* The hash table will store the tasks, the stack will be used for the undo method, the priority queue will be used to handle the tasks that have priority, and the queue for those that do not.
* The user should be shown the list of tasks (according to their priority) and reminders.

**Step 5. Evaluation and Selection of the Best Solution**

System Flow Diagram:



Criteria

* Criterion A. Aesthetics. The solution is pleasing to the user's eye:
  + [2] It has a good design.
  + [1] It's not pleasant.
* Criterion B. Completeness. It complies with the required functionalities.
  + [3] It fulfills all the functionalities that users need.
  + [2] It fulfills some, but not all, of the required functionalities.
  + [1] It does not meet the requested functionalities.
* Criterion C. Accuracy. The solution fulfills the functionalities using the relevant data structures.
  + [3] It uses all relevant data structures.
  + [2] It uses some, but not all, of them.
  + [1] It does not employ the relevant structures.
* Criterion D. Portability. The system can be used on a variety of devices.
  + [2] It can be accessed from many devices.
  + [1] It is very limited and can be accessed from few devices.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Criterion A | Criterion B | Criterion C | Criterion D | Total |
| Alternative 4. Development of a simple system in Java | 1 | 3 | 3 | 1 | 8 |

The selection that was decided to be taken was Alternative 4

Definition of all the TADs of the structures to be used

|  |
| --- |
| TAD Hash Table |
| A hash table is a data structure that implements a mapping function where is the set of keys and is the set of values. To do this, which is the behavior of a dictionary, two operations are performed: first a hash function where it is a finite set of positive integers. That is, the hash function converts the key into an integer. Second, in order to obtain the values that have been inserted into the hash table, a search is performed using the key associated with that value. The search is done by entering the key into the hash function and using the value returned by the function for.  ---  A hash table is a data structure that stores values in an array in the form of a dictionary. That is, when an object is entered into the hash table, it is done using a key and a value, the key is used to determine the position that the object will have in the array by means of a hash function, and what will be stored in that position will be the value of the object. |
| The invariant of a hash table is that for a k key, you always get the same value of the hash code, regardless of how many times the operation is performed.  Where k has a key, h the hash function, i a position in the array:  {inv:{ for all k it is true that when I enter the hash function I always get the same position}}  Can we consider the load factor as an invariant?  The load factor is defined as the number of items in the hash table divided by the total number of buckets in the table. In other words, it's the average number of items per bucket. If the load factor is too high, it means that there are too many items in the hash table relative to the number of buckets, which can lead to an increased number of collisions and therefore an increase in search time.  On the other hand, if the load factor is too low, it means that there are a lot of empty buckets, which can result in inefficient use of space.  Therefore, maintaining an optimal load factor is crucial to ensure a balance between time (speed of search, insert, and delete operations) and space (memory used by the hash table). |
|  |

|  |
| --- |
| TAD Priority Queue |
| ⟨abstract object⟩: PriorityQueue=⟨S (v, p), top, size⟩ |
| {Inv.: 0 ≤ n ∧ size(PriorityQueue) = n ∧ top = n ∧ ∀ (v\_1, p\_1), (v\_2, p\_2) ∈ S: p\_1 ≥ p\_2} |
| Constructor Operations  PriorityQueue  **Modifier opearations:**  Enque void  Dequeue  Front  **Analizing operations**  isEmpty boolean |

|  |
| --- |
| PriorityQueue  Builds an empty priorityQueue.  {pre: none}  {post: Queue q = Ø} |

|  |
| --- |
| Enqueue  An element is added to the queue with a priority (a numeric value).  {pre:PriorityQueue pq =⟨e\_ ("(v\_1, p\_1")) e\_ ("(v\_2, p\_" 2)) ... e\_ ("(v\_n, p\_" n)) ⟩ ∧element e or pq="Ø" ⟩ ∧element e}  {post: PriorityQueue pq = =⟨e\_ ("(v\_1, p\_1")) e\_ ("(v\_2, p\_" 2)) ... e\_ ("(v\_n, p\_" n)), e⟩ ∨q= ⟨e⟩} |

|  |
| --- |
| **Dequeue:**  Extracts the element in Queue q's front.  {pre: PriorityQueue pq i.e pq = ⟨ ⟩  {post:PriorityQueue pq==⟨ ⟩ } |

|  |
| --- |
| **isEmpty:**  **Determines if the PriorityQueue pq is empty or not.**  **{pre: PriorityQueue pq}**  **{post: true if s = Ø or false if s ≠ Ø}** |

|  |
| --- |
| TAD Stack |
| ⟨abstract object⟩:Stack=⟨⟨e\_ (1,) e\_ (2,) e\_ (3...,) e\_n ⟩, top⟩ |
| inv: Invariant of STACK: 0≤n ∧size (Stack)=n∧top=en } |
| **Operation Builder**  Stack  **Transaction Modifications**  Push void  Pop T  **Operations Analysis**  Size int  Top T  IsEmpty Boolean  Peek T |

|  |
| --- |
| Stack ()  Builds an empty stack.  {pre:}  {post: Stack s=Ø} |

|  |
| --- |
| push()  Adds the new element e to stack s.  {pre: Stack s = [Equation]}  {pos: Stack s = [Equation]} |

|  |
| --- |
| Pop  Extracts from the stack s, the most recently inserted element.  {pre: Stack s ≠ Ø}  {post: Stack s = [Equation]} |

|  |
| --- |
| Top  Recovers the value of the element on the top of the stack.  {pre: Stack s ≠ Ø}  {post: Element en } |

|  |
| --- |
| isEmpty  Determines if the stack s is empty or not.  {pre:Stack s}  {post: true if s = Ø or false if s ≠ Ø} |

|  |
| --- |
| TAD MaxHeap |
| Abstract Object of a MaxHeap:  - Elements: A MaxHeap contains a collection of elements e\_1, e\_2, e\_3, ..., e\_n. Each element is of a generic type T and is stored in an internal data structure, usually an array.  - Top: The "top" element of the MaxHeap is the highest priority element according to the order relationship defined by the comparison of the T-elements. |
| MaxHeap Invariant:  - 0 <= n: The number of elements in the MaxHeap is always greater than or equal to zero.  - size(MaxHeap) = n: The size of the MaxHeap is equal to the number of elements n stored in it.  - top = e\_n: The item at the top of the MaxHeap is equal to the last item added, ensuring that it is the highest priority item in the MaxHeap.  This invariant ensures that the MaxHeap stays properly sorted and that the element at the top is the highest priority based on the order relationship defined by T. |
| **Builders:**  **1. Stack() Builder:**  **Modifications to Operations:**  **2. Push(T element)-void:**  **3. Pop () - T:**  **Operations Analysis:**  **4. Size() - int:**  **5. Top () - T**  **6. IsEmpty() - boolean:**  **7. Peek() - T:** |

|  |
| --- |
| isEmpty operation in MaxHeap:  - The isEmpty operation determines whether the MaxHeap 'heap' is empty or not.  Precondition:  - The MaxHeap 'heap' is assumed to be defined and exists as input.  Insert (Push) method:  - Before inserting an element into the MaxHeap, check if the MaxHeap is empty using the isEmpty operation.  - isEmpty(heap) returns true if 'heap' contains no elements, i.e. if 'heap' is empty (size(heap) = 0).  - isEmpty(heap) returns false if 'heap' contains one or more elements, i.e. if 'heap' is not empty (size(heap) > 0).  extractMax (Pop) Method:  - Before extracting the maximum element from the MaxHeap, check if the MaxHeap is empty using the isEmpty operation.  - isEmpty(heap) returns true if 'heap' contains no elements, i.e. if 'heap' is empty (size(heap) = 0).  - isEmpty(heap) returns false if 'heap' contains one or more elements, i.e. if 'heap' is not empty (size(heap) > 0).  Size Method:  - The size method returns the number of items in the MaxHeap, but does not check whether the MaxHeap is empty or not.  heapMaximum (Top) method:  - The heapMaximum method returns the maximum element in the MaxHeap, but does not check whether the MaxHeap is empty or not.  Peek Method:  - The peek method returns the item on top of the MaxHeap without deleting it, but does not check whether the MaxHeap is empty or not. |

PSEUDOCODE:

Cola<T Class> Implements ICola<T>

Private Node<T> first

Private Node<T> last

Private Entire Size

Inner Class Node<T>

Private T Element

Private Node<T> Next

Queue Method ()

First <- Null

Last <- Void

Size <- 0

Empty method () -> Boolean

Return first == Null

Method Size () -> Integer

Return Size

Method QueryFront() -> T

If you empty () then

Throw Exception "Queue Overflow"

But

Return First.element

Glue Method (element: T)

New Node <- Create a new Node<T> with element and next as Null

If you empty () then

first <- NewNode

But

last.next <- NewNode

Last <- NewNode

Size <- Size + 1

Unglue Method () -> T

If you empty () then

Throw Exception "Queue Overflow"

But

Element <- First.element

First <- First.Next

Size <- Size - 1

If you empty () then

Last <- Void

End Yes

Return Item

Print Method () -> String

Message String <- ""

If first == Null Then

message <- "No Tasks Logged"

But

message <- PrintNode(first)

End Yes

Return Message

Print MethodNode(current: Node<T>) -> String

Message string <- current.element.ConvertAString()

If current.next ! = Null then

message <- message + " -> " + PrintNode(current.next)

End Yes

Return Message

Class Pair <K, T>:

Attributes

K key

T value

Methods

K getKey():

return key.

T getValue():

return value.

void setValue (T value):

value = value

void setKey (K key):

key = key

End.

Stack

Class Node <T>

Attributes

K key

Node <K> next

Methods

K getKey():

return key.

void setKey (K key):

key = key

getNext ()

return Node<K>

void setNext (Node<K> next)

next = next

End.

Class Stack:

Stack<K>:

Attributes

top: Node<K>

Constructor

Stack ():

top = null

Push method to add a new item to the top of the stack.

push (key: K):

node = new Node<K>(key)

if (top! = null) then

node. SetNext(top)

top = node

Pop method to delete the top item from the stack and return its key value.

pop (): K

if (isEmpty()) then

throw EmptyStackException.

pulled = top

top = top.getNext()

return pulled.getKey()

Top method to return the key value of the top item in the stack.

top (): K

if (isEmpty()) then

throw EmptyStackException

return top.getKey()

isEmpty method to check if the stack is empty.

isEmpty(): boolean

return top == null

Fin Stack<K>.

HASH TABLE:

Class arrayGen <T> // Also used for priorityQueue

Attributes

- [] array: T-type array

Methods:

- Constructor (Size: Int):

- Create a new size array

- isEmpty (index: int): boolean

- Return true if the item in the index position is null, otherwise return false

- setElement (index: int, element: T): void

- Assign the element element in the index position of the array

- getElement (index: integer): T

- Return the element to the index position of the array

- getSize(): int

- Return the array size

TableHash<K, V> implements ITablaHash<K, V>

Private HashNode Array<K, V> Table

Private HashNode<K, V> removed

Private Entire Size

KNUTH Decimal Constant <- (Square root of 5 - 1) / 2.0

Size == m

HashTable Method(Size: Integer)

Size <- Size

Table <- New HashNode<K, V> Arrays with Element Size

Method HashFunction(object: K) -> Integer

1. Get the String Representation of the Object

String info <- object. ConvertACadena()

2. Calculate the hash value using the hashcode method

Integer key <- object.hashCode()

If you key < 0 Then

Password <- Key \* (-1)

End Yes

3. Apply the multiplication method to calculate the hash rate

Integer hash <- Integer (Floor (size \* ((key \* KNUTH) %1))

Return hash

Add method (key: K, value: V) -> Boolean

Integer address <- Hash function(key)

Boolean result <- False

HashNode<K, V> newNode <- New HashNode<K, V> (key, value)

Collision Handling with Chaining

If table[address] == Null OR table[address] == deleted Then

table[address] <- newNode

Result <- True

But

HashNode<K, V> current <- table[address]

table[address] <- newNode

current.setPrevious(newNode)

newNode.setNext(current)

Result <- True

End Yes

Return Result

Find Method (Key: K) -> V

Integer address <- Hash function(key)

V Result <- Null

Position in the table

Yes table [address]! = Null then

result <- table[address].getValue()

End Yes

Return Result

Delete Method (Key: K)-> V

Integer address <- Hash function(key)

V Result <- Null

Position in the table

Yes table [address]! = Null then

HashNode<K, V> current <- table[address]

Result <- Current.GetValue()

If current.getNext()! = Null then

current <- current.getNext()

current.setPrevious(Null)

Table[Address] <- Current

But

table[address] <- removed

End Yes

End Yes

Return Result

Print Method () -> String

Message String <- ""

If table == Null Then

message <- "No Tasks Logged"

But

For index from 0 to size - 1

Yes table [index]! = Null then

message <- message + printNode(table[index])

End Yes

End To

End Yes

Return Message

Print MethodNode(current: HashNode<K, V>) -> String

Message String <- ""

V temp <- current.getValue()

Message <- Temp. ConvertACadena()

If current.getNext()! = Null then

message <- message + " -> " + printNode(current.getNext())

End Yes

Return Message

GetSize() -> Integer method

Return Size

GetTable() -> Method HashNode<K, V>

Return TableHeap

Tail<T Class>

Private Node<T> first

Private Node<T> last

Private Entire Size

Inner Class Node<T>

Private T Element

Private Node<T> Next

Queue Method ()

First <- Null

Last <- Void

Size <- 0

Vacuum Method () -> Boolean

Return first == Null

Method Size () -> Integer

Return Size

Method QueryFront() -> T

If empty () then

Throw Exception "Queue Overflow"

But

Return First.element

Glue Method (element: T)

New Node <- Create a new Node<T> with element and next as Null

If empty () then

first <- NewNode

But

last.next <- NewNode

Last <- NewNode

Size <- Size + 1

Unglue Method () -> T

If empty () then

Throw Exception "Queue Overflow"

But

Element <- First.element

First <- First.Next

Size <- Size - 1

If empty () then

Last <- Void

Return Item

Class MaxHeap<T extends Comparable<T>> implements IHeap<T>

Private ArrayList<T> heap

Private ArrayList<T> list

MaxHeap Method (list: ArrayList<T>)

List <- List

Heap <- list

Call buildMaxHeap()

Parent Method(Index: Integer) -> Integer

Return Integer (floor((index + 1) / 2) - 1)

Method left(index: Integer) -> Integer

Return 2 \* (index + 1) - 1

right(index: Integer) -> Integer method

Return 2 \* (index + 1)

Swap method (i: Integer, j: Integer)

T temp <- heap[i]

heap[i] <- heap[j]

heap[j] <- temp

Insert Method(Value: T)

Add the item at the end

Heap. Add(value)

Integer indexCurrent <- heap. Size()-1

Maintaining Mound Properties

heapIncreaseKey(currentindex)

extractMax() -> T method

Yes heap. Vacuum() then

Print "The mound is empty"

Return Null

End Yes

Yes heap. Size() == 1 Then

Retornar heap. Remove(0)

End Yes

Extract the Maximum Element from the Mound

T valueMax <- heap[0]

heap [0] <- heap. Remove(heap. Size()-1)

Maintaining Mound Properties

maxHeapify(0)

Return valueMax

maxHeapify method(CurrentIndex: Integer)

Integer indexLeft <- left(indexCurrent)

Integer indexRight <- right(CurrentIndex)

Integer IndexHighest <- IndexCurrent

If indexLeft < heap. Size() And heap[LeftIndex]. Compare With(heap[HigherIndex]) > 0 Then

indexHighest <- indexLeft

End Yes

If indexRight < heap. Size() And heap[rightIndex]. Compare With(heap[HigherIndex]) > 0 Then

IndexHighest <- IndexRight

End Yes

If CurrentIndex != IndexHigher, Then

swap (CurrentIndex, HigherIndex)

IndexCurrent <- IndexGreater

Descending on the mound

maxHeapify(CurrentIndex)

End Yes

buildMaxHeap() method

For i of (list. Size() / 2) up to 0 with pitch -1

maxHeapify(i)

End To

heapMaximum() -> T method

Retornar heap[0]

heapIncreaseKey method(index: Integer)

While index > 0 AND heap[parent(index)]. Compare With(heap[index]) < 0

swap (index, parent)

Index <- Parent(Index)

End While

print() -> String method

Message String <- ""

If heap == Null Then

Message <- "No Priority Tasks Stored"

But

For i from 0 to heap. Size()-1

Message <- Message + Heap[i]

End To

End Yes

Return Message

**Step 6. Preparation of Reports and Specifications**

\*\*Time Complexity Analysis for Operations:\*\*

1. \*\*Operation 'isEmpty'\*\*:

- The 'isEmpty' operation determines if the MaxHeap 'heap' is empty.

- Time Cost: O (1)

- Precondition: The MaxHeap 'heap' is assumed to be defined and exists as input.

2. \*\*Insert (Push) Method\*\*:

- Before inserting an element into the MaxHeap, check if the MaxHeap is empty using the 'isEmpty' operation.

- Time Cost: O (1)

- Precondition: It is assumed that the MaxHeap 'heap' is defined and exists as input.

- Post-condition:

- 'isEmpty(heap) = true' if 'heap' contains no elements, i.e. if 'heap' is empty (size(heap) = 0).

- 'isEmpty(heap) = false' if 'heap' contains one or more elements, i.e. if 'heap' is not empty (size(heap) > 0).

3. \*\*'extractMax' (Pop) method\*\*:

- Before extracting the maximum element from the MaxHeap, check if the MaxHeap is empty using the 'isEmpty' operation.

- Time Cost: O (1)

- Precondition: It is assumed that the MaxHeap 'heap' is defined and exists as input.

- Post-condition:

- 'isEmpty(heap) = true' if 'heap' contains no elements, i.e. if 'heap' is empty (size(heap) = 0).

- 'isEmpty(heap) = false' if 'heap' contains one or more elements, i.e. if 'heap' is not empty (size(heap) > 0).

4. \*\*Size Method\*\*:

- The 'size' method returns the number of elements in the MaxHeap.

- Time Cost: O (1)

- It doesn't check if the MaxHeap is empty or not.

5. \*\*'heapMaximum' (Top) method\*\*:

- The 'heapMaximum' method returns the maximum element in the MaxHeap.

- Time Cost: O (1)

- It doesn't check if the MaxHeap is empty or not.

6. \*\*Peek Method\*\*:

- The 'peek' method returns the item at the top of the MaxHeap without deleting it.

- Time Cost: O (1)

- It doesn't check if the MaxHeap is empty or not.

\*\*Summary:\*\*

In short, the operations 'isEmpty', 'insert', 'extractMax', 'size', 'heapMaximum' and 'peek' have a constant time of O(1), which means that the time required to execute these operations does not depend on the size of the MaxHeap. The total temporal complexity of the algorithm depends on the number of collisions that occur, which can lead to a worst-case scenario with O(n^2) complexity.

\*\*Time Complexity Analysis for Operations:\*\*

1. \*\*Operation 'isEmpty'\*\*:

- The 'isEmpty' operation determines if the MaxHeap 'heap' is empty.

- Time complexity: or (1)

- Precondition: It is assumed that the MaxHeap 'heap' is defined and exists as input.

2. \*\*Insert (Push) Method\*\*:

- Before inserting an element into the MaxHeap, check if the MaxHeap is empty using the 'isEmpty' operation.

- Time complexity: or (1)

- Precondition: It is assumed that the MaxHeap 'heap' is defined and exists as input.

- Post-condition:

- 'isEmpty(heap) = true' if 'heap' contains no elements, i.e. if 'heap' is empty (size(heap) = 0).

- 'isEmpty(heap) = false' if 'heap' contains one or more elements, i.e. if 'heap' is not empty (size(heap) > 0).

3. \*\*'extractMax' (Pop) method\*\*:

- Before extracting the maximum element from the MaxHeap, check if the MaxHeap is empty using the 'isEmpty' operation.

- Time complexity: or (1)

- Precondition: It is assumed that the MaxHeap 'heap' is defined and exists as input.

- Post-condition:

- 'isEmpty(heap) = true' if 'heap' contains no elements, i.e. if 'heap' is empty (size(heap) = 0).

- 'isEmpty(heap) = false' if 'heap' contains one or more elements, i.e. if 'heap' is not empty (size(heap) > 0).

4. \*\*Size Method\*\*:

- The 'size' method returns the number of elements in the MaxHeap.

- Time complexity: or (1)

- It doesn't check if the MaxHeap is empty or not.

5. \*\*'heapMaximum' (Top) method\*\*:

- The 'heapMaximum' method returns the maximum element in the MaxHeap.

- Time complexity: or (1)

- It doesn't check if the MaxHeap is empty or not.

6. \*\*Peek Method\*\*:

- The 'peek' method returns the item at the top of the MaxHeap without deleting it.

- Time complexity: or (1)

- It doesn't check if the MaxHeap is empty or not.

\*\*Summary:\*\*

In short, the operations 'isEmpty', 'insert', 'extractMax', 'size', 'heapMaximum' and 'peek' have a constant time of O(1), which means that the time required to execute these operations does not depend on the size of the MaxHeap. The total temporal complexity of the algorithm depends on the number of collisions that occur, which can lead to a worst-case scenario with O(n^2) complexity.

\*\*Time Complexity of the Algorithm: \*\*

The time complexity of the algorithm depends on the number of collisions that occur in the MaxHeap. In the worst-case scenario, the complexity can reach O(n^2), due to the creation of new temporary tables in case of collisions. In the best-case scenario, operations have a constant complexity of O(1), regardless of the size of the MaxHeap. Therefore, the time complexity of the algorithm can vary between O(1) and O(n^2) depending on the collisions.

\*\*Spatial Complexity of the Algorithm: \*\*

Spatial complexity analysis shows that the algorithm uses a constant amount of memory space, O(1), for local variables and non-recursive operations. In the case of recursive calls, additional stack space is used, contributing to a worst-case spatial complexity of O(n) due to recursive calls. The dominant term is O(n), which means that the memory space grows linearly with respect to the size of the MaxHeap.

**Test Case Design Class HasTable**

**Class HasTable**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective of the Test:** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Result** |
| HashTable | TestPush | Add multiple keys to the stack in different orders | . Push (key3); push(key1); push(key2) | The class is stored in order it moves up the stack |
| HashTable | TestPush | Try adding a duplicate key to the stack | Push(key1); push(key1); | The duplicate key is not stored in the stack |
| HashTable | TestPush | Try to add a null value to the stack | .push(null) | Stack Handles Null Values Properly |
| HashTable | TestTop | Get Major Key from an Empty Stack | Empty Stack | Returns a null value or throws an exception |
| HashTable | TestTop | Add multiple keys to the stack and verify that . TesTop() always return the G/L key without deleting | Add keys to the stack | Returns the major key without deleting it |
| HashTable | TesTop | Add Key with Negative Values to the Stack | .push(-5); .push(-10)  .push(-2) | Returns the most negative key |
| HashTable | TestEmpty | Check if the stack is empty after creating an empty stack using .isEmpty() | Empty Stack | Returns True |
| HashTable | TestEmpty | Check if the stack is empty after adding and deleting items using isEmpty() | Add and remove items from the stack | Returns flase |
| HashTable | TestEmpty | Perform a sequence, including adding, removing, and checking if the stack is empty | Multiple Operations on the Stack | Returns the correct value at every step |

Class Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective of the Test:** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Result** |
| Queue | TestisEmpty | Check if a newly created queue is empty | Newly created queue | Returns true |
| Queue | TestIsEmpty | Check if a queue with items is not empty | Add items to the queue | Returns false |
| Queue | TestIsEmpty | Check if a queue becomes empty after deleting all items | Add Items and Then Delete All of Them | Returns true after deleting all items |
| Queue | TestZise | Check the size of a newly created queue | Newly created queue | Returns 0 |
| Queue | TestZise | Check Queue Size After Adding Items | Add items to the queue | Returns the correct number of items in the queue |
| Queue | TestZise | Check the queue size after deleting the items | Add items and then delete some | Returns the correct number of items remaining in the queue |
| Queue | TestPeek | Check if peek returns the first item in an item queue | Add items to the queue | Return the first item without deleting it |
| Queue | TestPeek | Check if the first item hurts after deleting some items | Add items and then delete them | Returns the first remaining item in the queue |
|  |  |  |  |  |

**Class MaxHeap**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective of the Test:** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Result** |
| MaxHeap | TestparentIndex | Verify if the method returns the correct parent index when an even index is provided | Index=2 | Return 0 |
| MaxHeap | TestparentIndex | Verify if the method returns the correct inidce of the parent when an odd is provided | Index =3 | Return 1 |
| MaxHeap | TestparentIndex | Verify if the method returns the correct parent index when the root index is provided | Index =0 | Return 0 |
| MaxHeap | TestLeftIndex | Check if the method returns the correct index of the left child when an even index is provided | Index=2 | Return 3 |
| MaxHeap | TestLeftIndex | Verify if the method returns the correct index of the left child when an odd index is provided | Index=3 | Return 5 |
| MaxHeap | TestLeftIndex | Verify if the method returns the correct index of the left child when the root index is provided | Index =0 | Return 1 |
| MaxHeap | TestRightIndex | Verify if the method returns the correct index of the right child when an even index is provided | Index=2 | Return 4 |
| MaxHeap | TestRightIndex | Check if the method returns the correct index of the right child when odd is provided | Index=3 | Return 6 |
| MaxHeap | TestRightIndex | Verify if the method returns correct index of the right child when the index of the root is provided | Index=0 | Return 2 |
|  |  |  |  |  |

**Class Stack**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective of the Test:** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Result** |
| Stack | TestIsEmpty | Check if a newly created stack is empty | Newly Created Stack | Return true |
| Stack | TestIsEmpty | Check if a stack with items is not empty | Add items to the stack | Return false |
| Stack | TestIsEmpty | Check if a stack is empty after deleting all items | Add items and then delete them all | Return true |
| Stack | TestSize | Check the size of a newly created stack | Newly Created Stack | Return 0 |
| Stack | TestSize | Check the Stack Size After Deleting Items | Add items and then delete some | It should reflect the correct number of items remaining in the stack |
| Stack | TestSize | Check the size of the stack after removing the items | Add items and then delete some | It should reflect the correct number of items remaining in the stack |
| Stack | TestSize | Check the size of a newly created stack | Newly Created Stack | Return 0 |
| Stack | TestSize | Check the Stack Size After Adding Items | Add items to the stack | Return the correct number of items on the stack. |
| Stack | TestSize | Check the Stack Size After Deleting Items | Add items and then delete some | It should reflect the correct number of items remaining in the stack. |
| Stack | TestPush | Check if the method adds the stack correctly | Add an item to the stack | The stack contains the item added to the top |
| Stack | TestPush | Check if you can add a null value to the stack | Add a null value to the stack | The stack contains the null value at the top |
| Stack | TestPush | Check if it allows you to add multiple items to the stack and stores them correctly in order | Add Multiple Items to the Stack | The stack contains the added items in the correct order |
| Stack | TestPoP | Verify if you successfully delete the top item from a non-empty stack | Add items and then call pop | The stack does not contain the deleted item and its size decreases by one |
| Stack | TestPop | Check if it throws an exception when the empty stack is called | Empty pile | You must throw an exception  NoSuchElementException |
| Stack | TestPop | Check if it deletes all items from the stack and leaves it empty | Add items and then call pop | The stack is empty after deleting all items |
| Stack | TestPeek | Check if it throws an exception when called on an empty stack | Empty pile | You must throw an exception  NoSuchElementException |
| Stack | TestPeek | Verify if it correctly returns the top item of a non-empty stack | Add items to the stack and then call peek | The return value is the top element of the stack, and the stack is unchanged |
| Stack | TestPeek | Check if it returns the top item after removing some items from the stack | Add items and then call peek after removing some items | The return value is the remaining top item on the stack, and the non-stack is modified. |

**Step 7. Design Implementation**

Programming Language Implementation

Ready to deploy:

Subroutine Specification

|  |  |
| --- | --- |
| Name |  |
| Description |  |
| Entrance |  |
| Return |  |
|  |  |

Construction

Writing code in Java programming language