**THE DESIGN METHOD IN ENGINEERING**

**PHASE 1. IDENTIFYING THE PROBLEM**

**Problem identification:**

A group of software engineers called YZ Group wants to build a system that allows the management of tasks and reminders, due to the increasing need to organize daily responsibilities and stay on top of pending tasks in an increasingly busy world. Creating a task and reminder management system seeks to help people be more productive and meet their commitments effectively.

|  |  |
| --- | --- |
| **Customer** | YZ Group |
| **User** | Clients of YZ Group |
| **Functional requirements** | * FR 1. Create a task or reminder. * R.F 2. Deleted a task or reminder. * R.F 3. Modify a task or reminder. * R.F 4. Show the organize a task or reminder by priority. * R.F 5. Save changes made. * R.F 6. Undo changes. |
| **Context of the problem** | A group called YZ Group wants to build a system that allows the management of tasks and reminders, because they want the users to be able to write down in the system those important things that they want to remember. |
| **Non-functional requirements** | NFR1: Scalable Software |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 1. Create a task or reminder | | | |
| **Summary** | The system must allow the creation of tasks or reminder | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| id | String | | Every task or reminder has a unique identification code that is generated by the system |
| tittle | | String | Every task or reminder has a tittle. |
| description | | String | Every task or reminder can have a description without character. |
| dateLim | | Date | Some tasks and reminders have a deadline to change their status |
| priority | | int | Is the level of importance of the task or reminder |
| **General activities needed to obtain the results** | * Receive the attributes of the task or reminder. | | | |
| **Result or postcondition** | The task or reminder was successfully created | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
| errorMessage | String | | Show was an error during the creation of the task or reminder. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 2. Deleted a task or reminder | | | |
| **Summary** | The system must allow the elimination of tasks or reminder previously created. | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| id | String | | The unique identification must be used to search and delete the task or reminder |
| **General activities needed to obtain the results** | * Search the task or reminder to deleted. * Remove the task or reminder of the list of activities. | | | |
| **Result or postcondition** | The task or reminder was successfully deleted | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
| errorMessage | String | | Show it was an error during the elimination of the task or reminder. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 3. Modify a task or reminder | | | |
| **Summary** | The system must allow the modification of tasks or reminder | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| id | String | | The unique identification must be used to search and modify the task or reminder |
| change | | String | The changes that the user wants to do on the select task or reminder |
| **General activities needed to obtain the results** | * Search the task or reminder to modify. * Save the changes made to the task or reminder | | | |
| **Result or postcondition** | The task or reminder selected was successfully modify and save. | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
| errorMessage | String | | Show it was an error during the modifying of the task or reminder. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 4. Show the organize a task or reminder by priority | | | |
| **Summary** | The system must allow the view of the tasks or reminder organize by the priority | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
|  |  | |  |
| **General activities needed to obtain the results** | * Get the priority of every element on the list of task or reminder | | | |
| **Result or postcondition** | The tasks or reminders is show in an organized way | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
| msg | String | | Show the list of tasks or reminders organized by priority |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 5. Save changes made | | | |
| **Summary** | The system must save the changes, leaving first the last activity made for the user | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| action | String | | It’s used every time that the user realizes and action |
| **General activities needed to obtain the results** | * Every time that the user realizes an action get a description of what the user did. | | | |
| **Result or postcondition** | The actions made for the user are saved in the system to be used later. | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
|  |  | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 6. Undo changes | | | |
| **Summary** | The system must undo the last changes made | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| action | String | | It’s the last action made for the user |
| **General activities needed to obtain the results** | * Go to the list of activities made for the user. * Remove/deleted the last activity. | | | |
| **Result or postcondition** | The last action made for the user is deleted. | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
| errorMessage | String | | Show it was an error during the undo of the action. |

**PHASE 2. GATHERING OF THE NECESSARY INFORMATION**

**Import clarification to kept in mind:**

The client wants that the tasks/reminders be displayed in a certain order, in this case, by their priority, at a higher priority level they will be seen first. If the tasks/reminders have the same priority level, they will be displayed according to which one was created first. Since the tasks/reminders will be organized according to their priority and the first item will be the first out, you can see that the display of tasks/reminders follows the same behavior as a structure called a priority queue.

For the deletion of changes, it is known that only the last change made will be deleted, for this it is necessary that the last change made be the first to be deleted in the task/reminder management system, in this way it is known that elimination follows the same behavior as a structure call stack.

Every time that the user wants to modify or deleted a task/reminder, it should be search first, is for that reason that the client wishes to the task/reminder been save in a way that makes it easy and inexpensive for the system to search for them when its needed. Because the programming language asked to be used is Java, the team decided that it would be prudent to use a structure called Hash Table to fulfill the need to optimize task/reminder management.

**Needed information:**

***Hash Table***

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is O (1), which means that is not needed to go through the entire structure to find the element.

But what is defined as hashing, well, hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. An example of how hashing is used is in the universities, where each student is assigned a unique roll number.

The idea of hashing is to distribute entries uniformly across an array. Each element is assigned a key (converted key). Using the key, the algorithm (hash function) computes an index that suggests where an element can be found or inserted.

***Possible options of hash:***

**Open hashing:** Data can be stored in the form of chained lists within an infinite space, at least in theory. Although the keys are limited, chaining allows larger amounts of data to be processed.

**Closed hashing:** The number of keys is limited by the capacity of the table. If you try to save more data, a so-called overflow occurs. With each new scan, the table is polled for free positions in which to locate overflowed items.

All these concepts can be seen very abstract, so let's look at the following example:

Assume that you must store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab”, “defibs”}. To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to sum of ASCII values of characters added to the end their respective order in the string after which it is modulo with 2069, the ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively and as 20069 is a prime number, it will reduce the possibility of indexing different strings (collisions).

|  |  |  |
| --- | --- | --- |
| String | Hash function | Index |
| abcdef | (971 + 982 + 993 + 1004 + 1015 + 1026) %2069 | 38 |
| bcdefa | (981 + 992 + 1003 + 1014 + 1025 + 976) %2069 | 23 |
| cdefab | (991 + 1002 + 1013 + 1024 + 975 + 986) %2069 | 14 |
| defabc | (1001 + 1012 + 1023 + 974 + 985 + 996) %2069 | 11 |

The result will be the next:



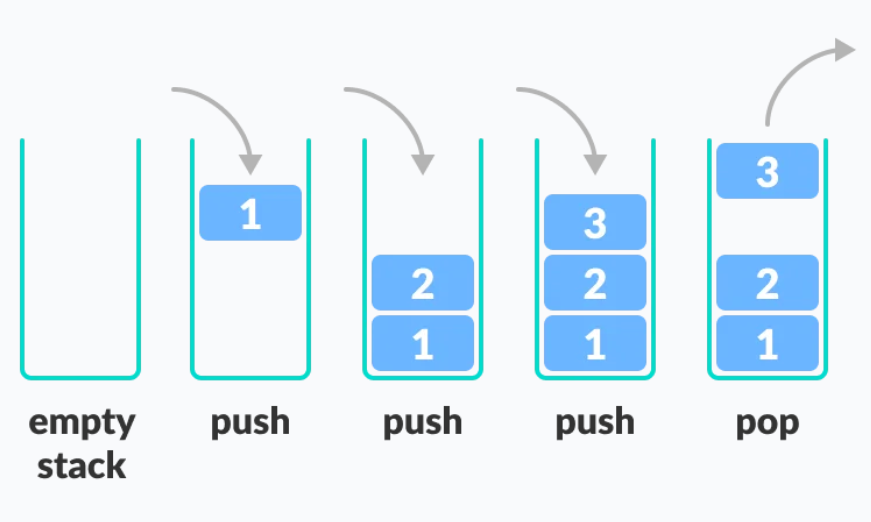
(Hacker Earth, 2016)

***Stack***

A stack is a linear data structure that follows the principle of Last in First Out (LIFO). This means the last element inserted inside the stack is removed first. You can think of the stack data structure as the pile of plates on top of another. Here, you can put a new plate on top or remove the top plate and, if you want the plate at the bottom, you must first remove all the plates on top. This is exactly how the stack data structure works.

There are some basic operations that allow us to perform different actions on a stack.

* Push: Add an element to the top of a stack.
* Pop: Remove an element from the top of a stack.
* Is-Empty: Check if the stack is empty.
* Peek: Get the value of the top element without removing it.



(Programiz, 2023)

***Queue***

A queue is a useful data structure in programming. It is like the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket. Queue follows the First in First Out (FIFO) rule - the item that goes in first is the item that comes out first. In programming terms, putting items in the queue is called enqueue, and removing items from the queue is called dequeue.

There are some basic operations that allow us to perform different actions on a queue:

* Enqueue: Add an element to the end of the queue.
* Dequeue: Remove an element from the front of the queue.
* Is-Empty: Check if the queue is empty.
* Peek: Get the value of the front of the queue without removing it.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

(Programiz, 2023)

***Priority Queue***

A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values. The hospital emergency is a real-life example of a priority queue. In this queue of patients, the patient with the most critical situation is the first in a queue, and the patient who doesn't need immediate medical attention will be the last.

In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value. For example, if you add an element with a high priority value to a priority queue, it may be inserted near the front of the queue, while an element with a low priority value may be inserted near the back.

So, a priority Queue is an extension of the queue with the following properties.

* Every item has a priority associated with it.
* An element with high priority is dequeued before an element with low priority.
* If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports the following operations:

* Insert: When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct place, then it will be compared with the parent node and the elements are swapped. The swapping process continues until all the elements are placed in the correct position.
* Deletion: It’s removed the root node from the queue, means, the one with maximum priority. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the order.
* Peek: This operation helps to return the maximum element from queue or the minimum element from queue without deleting the node from the priority queue.

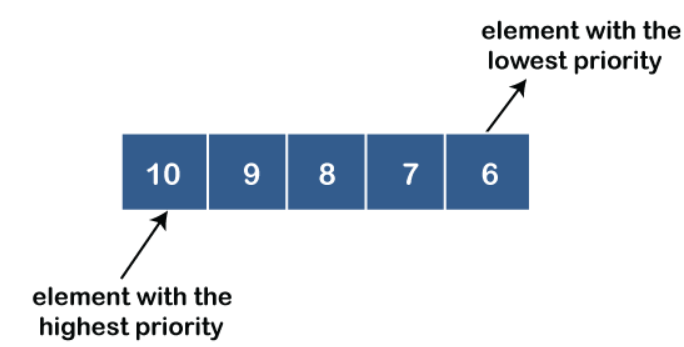
Types of Priority Queue:

* Ascending Order: As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in ascending order like 2, 6, 7, 10, 11. Here, 2 is the smallest number, therefore, it will get the highest priority in a priority queue and so when we dequeue from this priority queue, 2 will remove from the queue and dequeue returns 2.

Escala de tiempo

Descripción generada automáticamente

* Descending order: The root node is the element with the maximum priority. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue.



(Geeks for Geeks, 2023)

**PHASE 3: FINDING CREATIVE SOLUTIONS**

What structures can we use to solve the problem?

From the beginning there was an idea of the data structures to be implemented, so the solution revolved around these: stack, priority queue and hash tables. In phase two, each of them was defined, making it even easier to understand its application in relation to the problem. Even so, in the following phases there will be a detailed analysis of why these specific structures were chosen over others such as arrays or lists.

**Step 4. Transition from Ideas to Preliminary Designs:**

The hashtable will aim to store all information you are provided about the task and reminders. Priority queue and queue handle that information with respect to the priority given to tasks when they are assigned. And the batteries will undo the actions performed on the system.

**Tasks and reminders:**

**Hash table to store tasks and reminders:**

Efficiency: hash tables provide efficient data storage and retrieval. They are designed to store key-value pairs and, when properly implemented, offer a time complexity close to constant o (1) for common operations such as insertion, recovery, and elimination.

Contextual justification: in the context of a task and reminder management system, it is crucial to store and retrieve tasks efficiently. Users need to add, modify and retrieve tasks and reminders quickly. A well-implemented hash table ensures that tasks can be accessed through a unique identifier (key) with minimal computational cost.

**Pros:**

* quick search and access through passwords.
* efficient to store tasks and reminders, since you can directly access them with your key.

**Cons:**

* the performance of a hash table can be affected if there are key collisions.
* requires a good hash function to minimize collisions.

***Inefficiency of alternative approaches:***

**Use of arrangements to store tasks and reminders:**

Inefficiency: arrays have a fixed size, making it difficult to handle dynamically changing data. It would require resizing and frequent copying of arrangements to accommodate new tasks, leading to inefficiency.

 Contextual justification: the system involves adding, modifying and removing tasks on a regular basis. Arrays are not suitable for dynamic data management as they do not provide the flexibility to efficiently resize and reorganize the data structure when tasks are added, modified, or deleted.

**Pros:**

* quick access to elements through indexes.
* continuous space in memory.
* facilitates the implementation of heapsort functionality.

**Cons:**

* fixed size, which may require dynamic resizing.
* it is not efficient for inserting or removing elements in the middle of the structure.

**Using LinkedList to store tasks and reminders:**

Inefficiency: although linkedlist allows efficient insertion and deletion at the beginning and end of the list (o (1)), finding an item ina specific position has a time complexity of o (n), resulting in an inefficiency to quickly access tasks and key-reminders.

Contextual justification: in a task and reminder management system, efficiency in task search and retrieval is essential. Users need to quickly access their tasks and reminders, and password searching is a common operation. Linkedlist is not suitable for this specific operation.

**Pros:**

* Allows efficient insertion and deletion of items at the beginning and end of the list.
* Offers flexibility in list size.

**Con:**

* Slower search compared to hash tables to access items by key.
* Requires more memory space due to Pointers.

**Priority Queue for Task Priority Management:**

Efficiency: Priority queues are designed to efficiently manage elements according to their priority. Elements with higher priority are retrieved first, making them an ideal choice for task prioritization. Additionally, the time complexity for inserting and extracting elements from a PriorityQueue is O(log n), which is quite efficient.

Contextual Justification: The use of a priority queue allows for efficient organization of tasks based on their importance. Priority queues are ideal for adding tasks as they provide quick access to the most important tasks, ensuring they are handled first.

**Pros:**

* Automatic sorting of tasks by priority.
* Efficient for handling tasks with multiple priorities.
* Quick access to the highest-priority task.

**Conts:**

* Can be more complex to implement than other data structures.

**Queue to Non-Priority Task Management**

Efficiency: Queues are suitable for managing elements in a first-in, first-out (FIFO) order, making them efficient for handling tasks based on their creation order.

Contextual Justification: The "Non-priority" category needs to manage tasks based on their arrival order. A queue ensures that tasks are processed in the order they were added, following the FIFO principle.

***Alternative Approaches and Discarded Ones:***

**Use of Arrays to Manage Priority Tasks:**

Inefficiency: Using arrays to manage priority tasks would be inefficient in terms of time required for finding the most prioritized task. It would involve a sequential search to find the most prioritized task, resulting in a time complexity of O(n).

Contextual Justification: In a system that needs to assign and manage tasks according to their priority, using arrays to search for the most prioritized task would be impractical. Sequential searching through a set of tasks could be slow and not meet efficiency requirements.

**Pros:**

* Easy to implement.
* Continuous memory space.

**Cons:**

* Slow search for priority tasks (O(n)).
* Doesn't guarantee automatic prioritization order.

**Using a Simple Linked List to Manage Priority Tasks:**

Inefficiency: Simple linked lists are not efficient for managing priority tasks, especially if you need to quickly find the most prioritized task. Searching for the most prioritized task would have a time complexity of O(n), which is inefficient compared to a PriorityQueue.

Contextual Justification: In a task and reminder management system, efficiency in searching and handling priority tasks is crucial. Simple linked lists are not suitable for this task as they do not provide an efficient way to access the most prioritized task.

**Pros:**

* Allows efficient insertion and deletion of elements at the beginning and end of the list.
* Provides flexibility in terms of list size.

**Cons:**

* Slow search for priority tasks (O(n)).
* Does not guarantee automatic priority ordering.

**Undoing Actions:**

**Using Stacks (LIFO) to Undo Actions in the System:**

Efficiency: Stacks are highly efficient for the purpose of undoing actions in a task and reminder management system. The insertion and removal of elements in a stack have a time complexity of O(1), which means these operations are extremely fast.

Contextual Justification: Allowing users to undo actions is essential in a task and reminder management system. Stacks are suitable for this purpose as they provide an efficient way to track performed actions and revert them. Users can undo a previous action and return to a previous state quickly and easily.

**Pros:**

* Simple and efficient implementation for undoing actions.
* Efficient use of memory since only performed actions are stored.
* Works well for linear step-by-step undoing of actions.

**Cons:**

* Limited to linear step-by-step undoing of one action at a time. If a user wants to undo more than one action, they must perform multiple "undo" operations.
* Redoing previously undone actions may require additional implementation and, in some cases, may not be straightforward to achieve.

***Comparison with Alternative and Discarded Structures:***

**Using Doubly Linked Lists:**

Inefficiency: While doubly linked lists allow for efficient backward and forward traversal of the action history, they can be less memory-efficient compared to a stack. Each node in the doubly linked list must contain two additional pointers, one pointing to the previous node and another to the next node. This means that the data structure itself consumes more memory, which could be problematic in memory-constrained systems.

Contextual Justification: Similar to stacks, doubly linked lists are suitable for tracking undo actions in a task and reminder management system. They provide efficient access to previous and subsequent actions for backward and forward navigation through the action history. However, if users do not need the additional flexibility of more complex backward and forward navigation through the action history, the use of doubly linked lists may be unnecessary and add extra complexity to the system without a clear benefit.

**Pros:**

* Allows efficient backward and forward navigation through the action history.
* Does not limit the number of steps that can be undone, providing more flexibility to users.

**Cons:**

* Greater implementation complexity compared to stacks. More code is required to manage the additional pointers in the doubly linked list.
* Increased memory usage due to the additional pointers in each node of the list.

**Usage of Simple Linked List:**

Inefficiency: While a simple linked list could be a functional option, it is inefficient for undoing actions because it requires traversing the entire list from the first element to the last element, resulting in a time complexity of O(n). Additionally, the use of pointers adds an additional memory overhead.

Contextual Justification: The inefficiency of a simple linked list for undoing actions stems from the need to traverse the entire list, which is not suitable for a task and reminder management system where users are expected to quickly and efficiently undo actions. Users require fast access to the undo feature, and the simple linked list does not provide this efficiency.

**Pros:**

* Simple code implementation, as a simple linked list is a basic data structure.

**Cons:**

* Inefficient for undoing actions due to the O(n) time complexity of list traversal.
* Increased memory usage due to the presence of pointers in each list node.
* Does not provide fast and efficient access to the undo feature, which can be frustrating for users.

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

**Storage of Tasks and Reminders:**

Among our candidates, we have three structures characterized by their function of storing objects. First, we have Arrays, then a structure based on dictionary functionality, HashTable, and lastly, we have LinkedList.

**Criterion A. Time complexity: Element insertion time.**

* [2]O(1)
* [1]O(n)

**Criterion B. Temporal search complexity: Element search time.**

* [2]O(1)
* [1]O(n)

**Criterion C. Temporal complexity of removal: Element removal time.**

* [2]O(1)
* [1]O(n)

**DE criterion. LookUp Time Complexity: LookUp time of elements.**

* [2]O(1)
* [1]O(n)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Criterion A | Criterion B | Criterion C | Criterion D | Total |
| Arrays | 1 | 1 | 1 | 2 | 5 |
| Hash Table | 2 | 2 | 2 | 2 | **8** |
| Double Linked List | 2 | 1 | 1 | 2 | 6 |

**Task Management:**

**Efficiency in Retrieving Priority Tasks:**

* [2]O(log n)
* [1]O(n)

**Automation of Priority Order:**

* [2] The data structure automates the process of sorting tasks by priority
* [1] stores elements in a contiguous block of memory

**Flexibility in Terms of List Size:**

* [2] Can adapt to accommodate a variable number of tasks or elements.
* [1] Have a fixed size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Criterion A | Criterion B | Criterion C | Total |
| Queue | 2 | 2 | 2 | **6** |
| Arrays | 1 | 1 | 1 | 3 |
| LinkedList | 1 | 1 | 2 | 4 |

**Efficiency in Undoing Actions:**

[2] The data structure efficiently supports undoing actions with a low time complexity.

[1] The data structure is less efficient in supporting the undo action with higher time complexity.

**Memory Usage:**

[2] The data structure uses memory efficiently for storing undo actions.

[1] The data structure may consume more memory for storing undo actions.

**Implementation Complexity:**

[2] Implementation of the data structure and undo mechanism is straightforward.

[1] Implementation of the data structure and undo mechanism is complex and may require additional effort.

|  | Criterion A | Criterion B | Criterion C | Total |
| --- | --- | --- | --- | --- |
| Stack | 2 | 2 | 2 | 6 |
| Double Linked List | 2 | 1 | 2 | 5 |
| LinkedList | 1 | 2 | 2 | 5 |

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

|  |  |  |
| --- | --- | --- |
| **TAD Stack** | | |
| A stack is a list of elements in which the last object in is the first out | | |
| *inv*:   * Each new element is added to the top of the stack * The last element added is the firts to be removed | | |
| create |  | * Stack |
| push | Stack x element | * Stack |
| pop | Stack | * Stack |
| peek | Stack | * Stack |
| isEmpty | Stack | * boolean |

|  |
| --- |
| create |
| Constructor |
| Build a new stack without elements |
| precondition: |
| postcondition: A new empty stack is created |

|  |
| --- |
| push |
| Modifier |
| Add an element to a stack |
| precondition: There must be a stack previously created |
| postcondition: A new element is add/save in the stack |

|  |
| --- |
| pop |
| Modifier |
| Deleted the element at the top of the stack |
| precondition: There must be a stack previously created |
| postcondition: The top element is deleted |

|  |
| --- |
| peek |
| Analyzer |
| Show the top element of the stack without modify or deleted it |
| precondition: There must be a stack previously created |
| postcondition: The top element is returned |

|  |
| --- |
| isEmpty |
| Analyzer |
| Show if the stack contains elements or not |
| precondition: There must be a stack previously created |
| postcondition: The stated of the stack is show |

|  |  |  |
| --- | --- | --- |
| **TAD Queue** | | |
| A queue is a list of elements in which the first object in is the first out | | |
| *inv*:   * Each new element is added to the end of the queue * The first element added is the first to be removed | | |
| create |  | * Queue |
| enqueue | Queue x element | * Queue |
| dequeue | Queue | * Queue |
| peek | Queue | * Queue |
| isEmpty | Queue | * boolean |

|  |
| --- |
| create |
| Constructor |
| Build a new queue without elements |
| precondition: |
| postcondition: A new empty queue is created |

|  |
| --- |
| enqueue |
| Modifier |
| Add an element to the end of a queue |
| precondition: There must be a queue previously created |
| postcondition: A new element is added in the queue |

|  |
| --- |
| dequeue |
| Modifier |
| Deleted the firts element of the queue |
| precondition: There must be a queue previously created |
| postcondition: The firts element is deleted |

|  |
| --- |
| peek |
| Analyzer |
| Show the firts element of the queue without modify or deleted it |
| precondition: There must be a queue previously created |
| postcondition: The firts element is returned |

|  |
| --- |
| isEmpty |
| Analyzer |
| Show if the queue contains elements or not |
| precondition: There must be a queue previously created |
| postcondition: The stated of the queue is show |

|  |  |  |
| --- | --- | --- |
| **TAD Hash Table** | | |
| It is a structure where an ***n*** number of elements is stored that needs to be added and searched efficiently. Each element has a key and a value. | | |
| *inv*:   * An element must be added and searched and deleted using just the key. | | |
| create |  | * HashTable |
| hash | HashTable x key | * HashTable |
| insert | HashTable x key x element | * HashTable |
| search | HashTable x key | * element |
| deleted | HashTable x key | * element |
| isEmpty | HashTable | * boolean |

|  |
| --- |
| create |
| Constructor |
| Build a new hash table without elements |
| precondition: |
| postcondition: A new empty hash table is created |

|  |
| --- |
| hash |
| Modifier |
| Transforms the key in a position of the hash table |
| precondition: There must be a hash table previously created |
| postcondition: A position is determined |

|  |
| --- |
| insert |
| Modifier |
| Add a value in the hash table in the position determined for the hash function |
| precondition: There must be a hash table previously created |
| postcondition: A new value is added on the hash table |

|  |
| --- |
| search |
| Analyzer |
| Found an element in the hash table using the key |
| precondition: There must be a hash table previously created |
| postcondition: Return the element in the position determined for the hash function on the hash table |

|  |
| --- |
| deleted |
| Modifier |
| Deleted an element in the position determined for the hash function of the hash table |
| precondition: There must be a hash table previously created |
| postcondition: The element is deleted |

|  |  |  |
| --- | --- | --- |
| **TAD Priority Queue** | | |
| It priority queue is a list of elements in which the first object in is the first out, but, the elements in the priority queue are organized according to the level of priority they have. | | |
| *inv*:   * The highest priority element is the smallest if it is a minimum priority queue or the biggest if it is a maximum priority queue. | | |
| create |  | * PriorityQueue |
| build\_Heap | PriorityQueue x elements | * PriorityQueue |
| heapify | PriorityQueue x element | * PriorityQueue |
| maximun | PriorityQueue | * element |
| deleted | PriorityQueue | * element |
| isEmpty | PriorityQueue | * boolean |

|  |
| --- |
| create |
| Constructor |
| Build a new priority queue without elements |
| precondition: |
| postcondition: A new empty priority queue is created |

|  |
| --- |
| build\_Heap |
| Modifier |
| Add and organized all the elements given to an empty priority queue |
| precondition: There must be a priority queue previously created |
| postcondition: A organized priority queue with elements |

|  |
| --- |
| heapify |
| Modifier |
| Add one element to the priority queue in the position determined by the priority of the element |
| precondition: There must be a priority queue previously created |
| postcondition: A new value is added on the priority queue |

|  |
| --- |
| maximum |
| Analyzer |
| Show the first element of the priority queue without modify or deleted it |
| precondition: There must be a priority queue previously created |
| postcondition: The first element is returned |

|  |
| --- |
| deleted |
| Modifier |
| Deleted an element in the position determined for the hash function of the hash table |
| precondition: There must be a hash table previously created |
| postcondition: The element is deleted |