**THE DESIGN METHOD IN ENGINEERING**

**Problematic context**

In everyday life, people face an increasingly hectic and demanding environment, characterized by an overwhelming amount of responsibilities, tasks and reminders that they must manage. The multiplicity of work roles and commitments, as well as the constant digital interconnection, gives rise to an increasing complexity in the organization of daily activities, which generates a great difficulty for people to keep abreast of their activities and commitments.

To address this problem, a task and reminder management system has been granted, this must be integrated as a comprehensive solution designed to provide an effective solution so that the user can manage and organize their responsibilities. In this current context, the Task and Reminder Management System is presented as a very valuable technological tool to achieve a balance between work and personal life. In the digital age, managing tasks and reminders has become increasingly challenging due to the vast amount of information and distractions we constantly face. Therefore, this system is a great ally to help us stay organized and remember our important commitments by giving them a higher priority.

**PHASE 1: IDENTIFYING THE PROBLEM**

Problem: Lack of an efficient system for users to manage their pending tasks and reminders.

Needs:

* Organization of tasks and reminders.
* Prioritization of tasks.
* Ability to undo actions.
* Design a user interface.
* Quick access to information.
* Registration of actions

|  |  |
| --- | --- |
| CUSTOMER |  |
| USER |  |
| FUNCTIONAL REQUIREMENTS | 1. **Storage of tasks and reminders** 2. **User interface** 3. **Priority Management** 4. **Undo actions** |
| CONTEXT OF THE PROBLEM | It seeks to develop a system that allows users to register and organize tasks with detailed information, prioritize them and have an undo option. This will facilitate the effective organization and monitoring of their daily activities, ensuring greater efficiency in the management of their responsibilities and personal objectives. |
| NON-FUNCTIONAL REQUIREMENTS | 1. **Yield:** 2. **Usability:** 3. **Safety:** 4. **Scalability:** 5. **Availability:** 6. **Backup and Recovery:** 7. **Compatibility:** 8. **Documentation:** |

**Functional requirements analysis table (Note: One table for each functional requirement)**

| Name or identifier | **RF1: Storage of tasks and reminders** | | |
| --- | --- | --- | --- |
| Summary | **The system must allow users to store tasks and reminders in a hash table.** | | |
| Tickets | Entry name | Data type | Selection or repetition condition |
| **title** | **String** | **Character string** |
| **dateLim** | **Date** | **dd/mm/yy** |
| **priority** | **String** | **Character string** |
| Result or postcondition | The task or reminder is stored in the hash table. | | |

| Name or identifier | **RF2: User Interface** | | |
| --- | --- | --- | --- |
| Summary | **The system should provide a user interface that allows users to add, modify, delete, and view an ordered list of tasks and reminders.** | | |
| Tickets | Entry name | Data type | Selection or repetition condition |
| **Action** | **String** | **User actions (for example, "Add Task", "Modify Task", "Delete Task")** |
| Result or postcondition | User actions are performed successfully and the view of the task list and reminders is updated. | | |
| Outputs | Exit name | Data type | Selection or repetition condition |
| task | Agenda |  |
| reminder | Agenda |  |
| .msg | String | Confirmation or error messages for actions performed by the user. |

| Name or identifier | **RF3: Priority Management** | | |
| --- | --- | --- | --- |
| Summary | **The system should allow users to manage tasks in two categories: "Priority" and "Non-priority".** | | |
| Tickets | Entry name | Data type | Selection or repetition condition |
| **selectPriority** | **String** | **"Priority", "Not priority"** |
| Result or postcondition | Tasks are organized into categories selected by the user | | |
| Outputs | Entry name | Data type | Selection or repetition condition |
| listTasks | String | View of the task list organized according to selected categories |

| Name or identifier | **RF4: Undo Actions** | | |
| --- | --- | --- | --- |
| Summary | **The system must allow users to undo the last action performed on the system.** | | |
| Tickets | Entry name | Data type | Selection or repetition condition |
| **Option** | **String** | **Selection of the "Undo" option by the user** |
| Result or postcondition | The last action performed by the user is successfully rolled back | | |
| Outputs | Entry name | Data type | Selection or repetition condition |
| .msg | String | Confirmation or error messages for the undo action |

**PHASE 2: GATHERING THE NECESSARY INFORMATION**

**Battery**

A stack consists of data and a stop, the data are a set of elements, generally of the same type, ordered implicitly and accessible from a single point, the stop is the indicator of the position of the last inserted element; results in LIFO ordering. (Cantador, s.f.)

**LIFO: Last-In, First-Out**

The LIFO stack management pattern or LIFO algorithm is based on the format that indicates that the last request to enter must be the first to be attended and exited. This is because it is assumed that the requests that have been in queue longer are gone, so attending to them is no longer a priority. On the other hand, new requests are understood by the algorithm as options with greater probabilities of being served by the issuer. (keepcoding., s.f.)

**Tail**

The queue data type is a data structure that organizes data as follows: From a memory address, data is successively stored as if it were an ordered collection of items. (sedici.unlp.edu.ar, s.f.), this works with FIFO sorting.

**FIFO: First-In, First-Out**

The FIFO pattern or FIFO program, on the other hand, is based on the principle that the request that arrives first must be the one that is attended and leaves first, while the newest request is attended to at the end of the queue. (keepcoding., s.f.)

**Hash table**

A hash table or hash map is a data structure that associates keys or keys with values. The main operation that it supports efficiently is the search: it allows access to the elements (telephone and address, for example) stored from a key generated using the name, account number or id. It works by transforming the key with a hash function into a hash, a number that the hash table uses to locate the desired value. (Ingeneriaudb, s.f.)

**Priority queue**

It is one of the classes that implements this interface and sorts the elements based on their natural order, as specified by the method compared the elements**Comparable**, or by using an object**Comparator**which is supplied through the builder. This class provides functionality that allows insertions in order and deletions of the front (decollate normally). (pcresumen, s.f.)

**TAD**

A TAD is an abstract type of data (TAD) is a collection of properties and operations that are defined by a specification that is independent of any representation. Abstraction focuses on the independence of representation. This allows the programmer to modify the representation of the TAD without affecting its use. (openaccess, s.f.)

**User interface**

The user interface is the means by which a person uses a software application or hardware device. That is, the program includes graphic controls that optimize the user experience when using a mouse or keyboard, which makes it possible to interact with the processors to perform a job. (Lenis, s.f.)

**Task**

The word task is used to refer to an activity or practice that is demanded for various reasons such as academic, work, domestic, etc. Generally, these activities or practices are framed in a specific time or situation. In other words, tasks are works or works to be developed that are delimited by certain rules (time, space, forms, media, etc.). (conceptodefinicion, s.f.)

**PHASE 3: FINDING CREATIVE SOLUTIONS**

In this problem, solutions can be established such as:

Brainstorming:

* Create classes that represent tasks and reminders, and provide methods for manipulating them.
* Implements logic to sort tasks by deadline or priority.
* Design an intuitive and engaging user interface that allows users to add, modify, and delete tasks and reminders efficiently.
* Implement an advanced search feature that allows users to quickly find specific tasks on their list.
* Incorporate the option to assign tasks to other users or collaborators for shared management.
* Review and optimize the hash table structure to ensure fast and efficient access to tasks and reminders.
* Evaluate the user experience in the user interface and make sure it is friendly and easy to navigate.
* Perform and optimize the hash table structure to ensure fast and efficient access to tasks and reminders.
* Evaluate the user experience in the user interface and make sure it is friendly and easy to navigate.
* Create an action stack: Implements a stack-like data structure (LIFO) to track user actions.
* Log Actions: Each time the user performs an action (add, modify, or delete a task), it records the action in the stack with details such as the type of action and details of the affected task.
* Undo method: Implements a method that allows you to undo the last action performed by the user. This method will unstack the last action from the stack and reverse the corresponding action based on the information stored in the stack.
* Using the Undo Method: In the user interface, it provides users with the option to "Undo". When they select this option, it calls the undo() method, which will restore the last action performed by the user.
* Implement unit tests for each system component.
* Perform integration tests to ensure that all parts of the system work properly together.

**PHASE 4: TRANSITION FROM IDEAS TO PRELIMINARY DESIGNS**

In this phase, the preliminary designs of the Task and Reminder Management System will be carried out, based on the creative solutions identified in Phase 3:

1. **Representative Classes:**
   * Preliminary Design:
     + Create a **Task class**  with attributes such as **id**, title**, description**, deadlinedate**, priority**  **, etc.**
     + Create a **Reminder class**  similar to the **Task class**  with relevant attributes.
   * Preliminary Methods:
     + In both classes, include methods for accessing and modifying attributes, such as getters and setters.
2. **Task Ordering:**
   * Preliminary Design:
     + Implement a sorting service that allows users to choose between sorting by deadline or priority.
     + Use sorting algorithms such as heapsort or quicksort to perform sorting.
3. **User Interface (UI):**
   * Preliminary Design:
     + Design a window-friendly user interface for adding, modifying, and deleting tasks and reminders.
     + Include buttons and input fields to interact with tasks and reminders.
     + Display a list of tasks and reminders in a dynamically updated table.
4. **Search:**
   * Preliminary Design:
     + Add a search bar that allows users to search for tasks by keywords, dates, priority, etc.
     + Implement an efficient search algorithm that returns relevant results.
5. **Task Assignment:**
   * Preliminary Design:
     + Allow users to assign tasks to other users or collaborators.
     + Design a function that allows you to select recipients and assign tasks through a drop-down list or input field.
6. **Hash Table Optimization:**
   * Preliminary Design:
     + Evaluate and adjust the hash function used in the table to ensure quick and efficient access to tasks and reminders.
     + Consider collision resolution techniques, such as chaining or quadratic scattering.
7. **User Experience:** 
   * Preliminary Design:
     + Perform usability testing to ensure the user interface is intuitive and easy to use.
     + Get feedback from users and make adjustments to the interface as needed.
8. **Stack of Actions and Undo:**
   * Preliminary Design:
     + Implement a **Stack** class that supports stack()  **and** unstack() **operations.**
     + Record user actions (such as adding, modifying, or deleting tasks) in the stack.
     + Design an **undo() method**  that unstacks the last action and reverts it based on the stored information.
9. **Priority Management:**

Preliminary Design:

* It implements a priority system that allows users to categorize their tasks into two levels: "Priority" and "Non-priority".
* Add a new task, provides an option for the user to select their priority level.
* Use a priority queue to organize priority tasks. When a user adds a new priority task, it is inserted into the priority queue based on its level of importance, allowing important tasks to be handled first.
* For non-priority tasks, these will be managed on a first-come, first-served basis using a FIFO (First-In-First-Out) data structure.

1. **Testing and validation:**
   * Preliminary Design:
     + Create a set of unit tests to verify the operation of classes and methods.
     + Design integration tests that simulate the interaction between system components and verify their cohesion.

**PHASE 5: EVALUATION AND SELECTION OF THE BEST SOLUTION**

**Definition of Evaluation Criteria**

To evaluate and select the best solution, we will define criteria based on the needs and requirements identified in the previous phases:

**Criterion A. Accuracy of the solution:**

* [2] Accuracy (precise solution preferred)
* [1] Acceptable approximation

**Criterion B. Efficiency:**

* [4] Constant efficiency
* [3] Efficiency greater than constant
* [2] Logarithmic efficiency
* [1] Linear efficiency

**Criterion C. Completeness:**

* [3] All possible solutions
* [2] Most, but not all, solutions
* [1] Only one solution or none

**Criterion D. Ease of algorithmic implementation:**

* [2] Compatible with the basic arithmetic operations of modern computer equipment
* [1] Not fully compatible with the basic arithmetic operations of modern computer equipment

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Criterion A (Precision)** | **Criterion B (Efficiency)** | **Criterion C (completeness)** | **Criterion D (Ease of Implementation)** | **Total** |
| **Representative calses** | 1 | 3 | 2 | 2 | 8 |
| **Task sorting** | 2 | 3 | 3 | 2 | 10 |
| **User interface (UI)** | 2 | 3 | 3 | 2 | 10 |
| **Search** | 2 | 3 | 3 | 2 | 10 |
| **Task assignment** | 2 | 3 | 3 | 2 | 10 |
| **Hash Table Optimization** | 2 | 3 | 3 | 2 | 10 |
| **User Experience** | 1 | 3 | 2 | 2 | 8 |
| **Action stacks and undo** | 1 | 3 | 2 | 2 | 8 |
| **Priority Management** | 2 | 4 | 2 | 2 | 10 |
| **Testing and Validation** | 2 | 3 | 3 | 2 | 10 |

All proposed solutions appear to be viable and obtain a total score of 10 according to the defined criteria. This suggests that each solution has its merits and can be considered based on your project's specific needs and individual priorities.

**PHASE 6. PREPARATION OF REPORTS AND SPECIFICATIONS**

Problem: Implement a system that helps people create, sort, and control their important tasks and reminders.

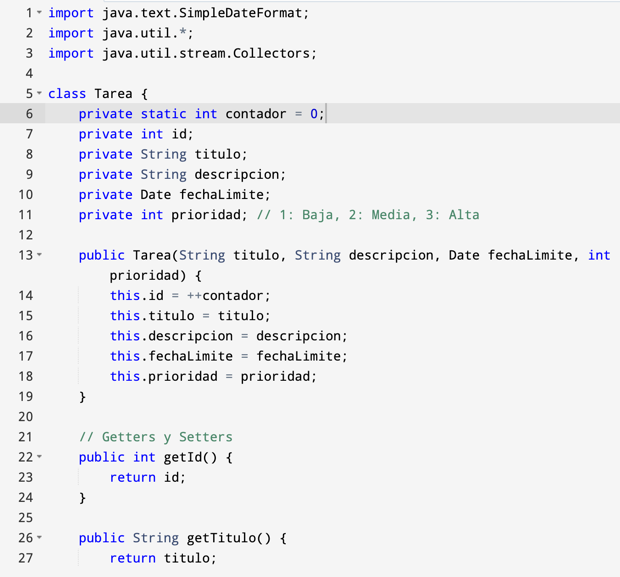
Tickets:

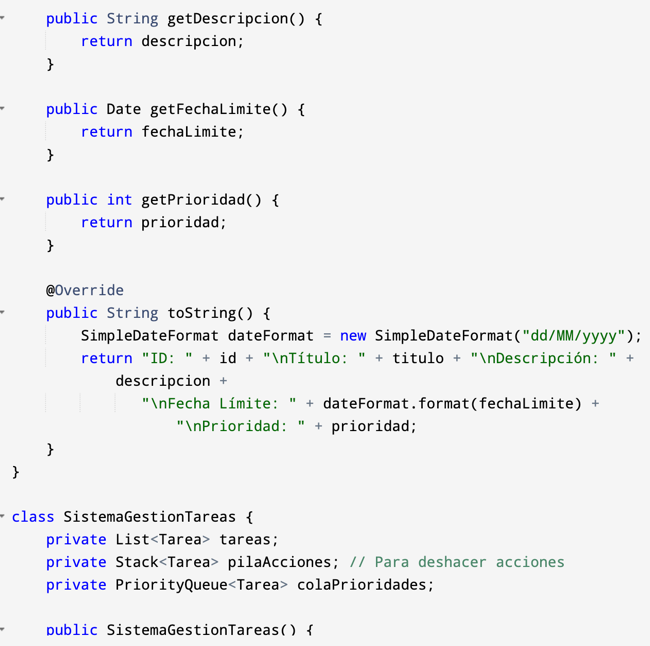
* Task or reminder.
* Optional description
* Priority (high, medium, low).
* Instructions for adding, editing, or deleting tasks and reminders via text commands.
* Queries to search for specific tasks by title, date, priority, etc.

Outputs:

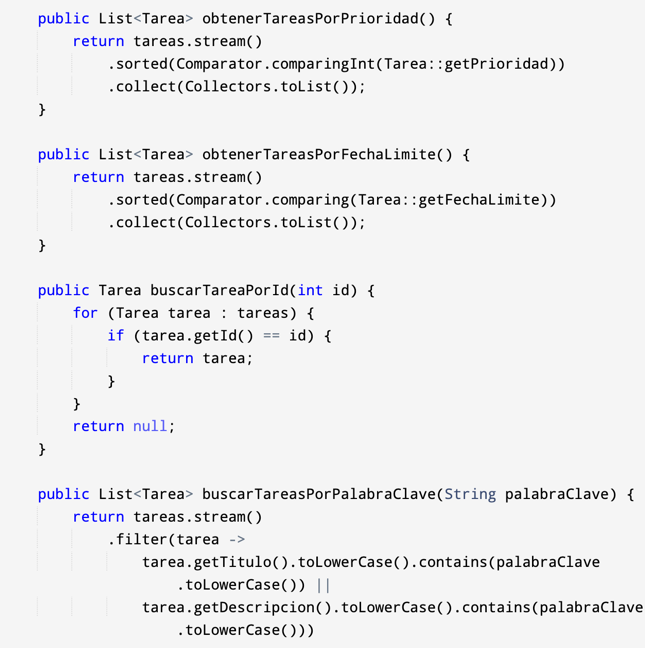
* To-do list and reminders, prioritized.
* Details of a specific task or reminder, including its title, description.
* Saving data to a database or file for persistence.
* Error messages that inform the user about problems, such as incorrect input or problems saving data.
* Confirmation of undo and redo actions when available.

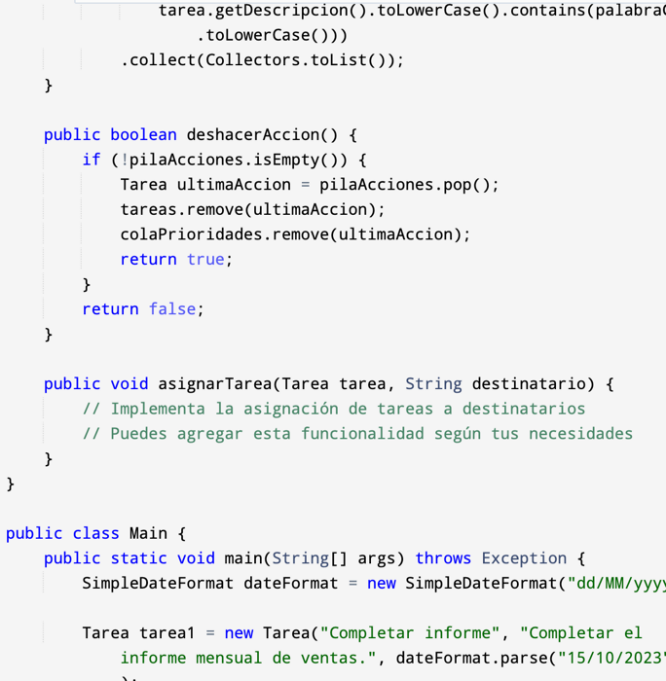
Preliminary design:







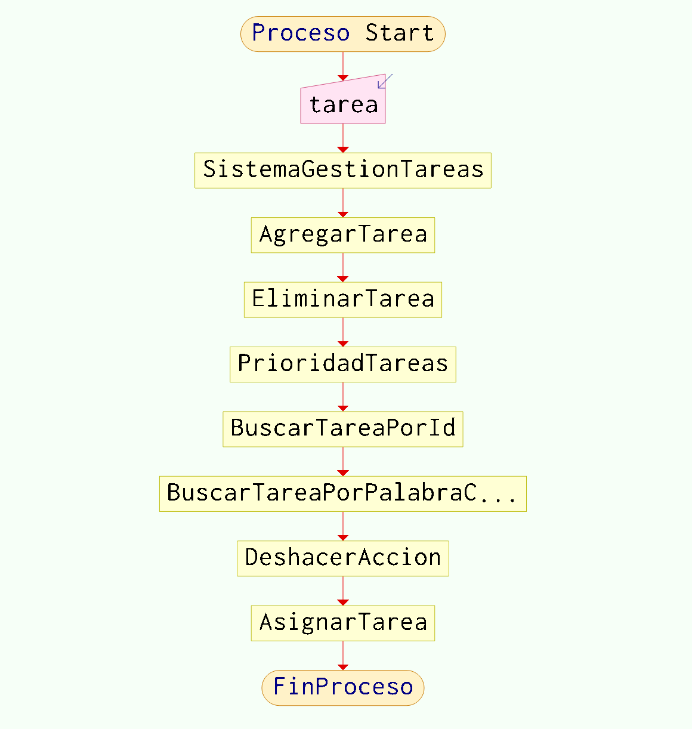








Flowchart:



|  |
| --- |
| TAD <Max priority queue > |
| Max priority queue={size,comparator} |
| Inv: {comparator(a,b)= True} |
| Primitive Operations:  Createpriorityqueue(size): ->priorityqueue  enqueue(data) : priorityqueuexnodo -> void  Dequeue(): priorityqueue->Node  peek (): -> node  size(): -> data  Clear (): -> void |

**Max Priority Queque**

|  |
| --- |
| Createpriorityqueue(Size)  “Creates a new priority queque”  {pre: Size}  {pos: print priorityqueque data} |

|  |
| --- |
| enqueue(date)  “Adds data to queque”  {pre: data}  {pos: print priorityqueque data with enquque data } |

|  |
| --- |
| Dequeue()  “Removes data from queque ”  {for: !=null}  {pos: print priorityqueque data without Dequeque data } |

|  |
| --- |
| peek ()  "print quequeque"  {for: !=null}  {POS: Prints Queque} |

|  |
| --- |
| size()  “Determines queque size”  {for: !=null}  {pos: queque size} |

|  |
| --- |
| TAD <Stack > |
| Stack = {size, comparator} |
| Inv: {comparator(stack, element1, element2) = True} |
| Primitive Operations:  **CreateStack(size)**: ->stack  **push(data)**: Adds an item to the stack.  **pop()**: Deletes and returns the item at the top of the stack.  **peek()**: -> node  **size()**: -> data  **clear()**: -> void |

**Stack**

|  |
| --- |
| **CreateStack(size)**  "Create a new stack."  {pre: Size}  {pos: "Print the empty stack structure.} |

|  |
| --- |
| **push(data)**  “"Add data to the stack."”  {pre: data}  {pos "Print the stack structure with data added to the top."} |

|  |
| --- |
| **pop()**  “Removes and returns the element at the top of the stack”  {for: !=null}  {pos: "Print the stack structure without the removed element.} |

|  |
| --- |
| peek ()  “Returns the element at the top of the stack without removing it.”  {for: !=null}  {pos: print the element at the top of the stack} |

|  |
| --- |
| size()  “Determines stack size”  {for: N/A}  {pos: “Print the current stack size”} |

|  |
| --- |
| **clear()**  “empty the stack”  {for: !=null  { pos: “Print the empty stack structure.” } |

|  |
| --- |
| THEN <Queue > |
| Queue={add,poll,size} |
| Inv: {comparator(a,b)= True ^ (Q={x1,x2,x3,x4,x5... xn} ^ Q.poll()=x1)} |
| Primitive Operations:  CreateQueue(): ->Queue  add(Node) : add. Node -> void  poll(): Queue->Node  size(): -> data  isEmpty (): -> boolean |

**Queue:**

|  |
| --- |
| CreateHashTable(add,poll,size)  “Creates a new queue”  {pre add,poll,size }  {pos: String “queue created”} |

|  |
| --- |
| add(Node)  “adds a node to the queue”  {pre: Node!=null y queue !=null}  {pos: String “added”} |

|  |
| --- |
| poll ()  “takes the first element out of the queue”  {pre: queue ¡= null}  {pos: Node} |

|  |
| --- |
| size ()  “find and return the queue size”  {pre: queue !=null }  {pos: prints size} |

|  |
| --- |
| isEmpty()  “determines if the queue is empty”  {pre: queue! = null}  {pos: boolean} |

Hashtable

|  |  |  |
| --- | --- | --- |
| **HashTable = [L0, L1, L2, ..., LN-1], where Li is a linked list containing all tuples (ki, vi) such that h(ki) = i, for all i = 0, 1, 2, ..., N-1.** | | |
| ∀ i j ∈ [0, N-1], i ≠ j ⇒ (ki ≠ kj) ∨ (h(ki) ≠ h(kj)) ∨ (ki = kj ∧ nodes[h(ki)] contains (ki, vi) ∧ nodes[h (kj)] contains (kj, vj)) | | |
| **Primitive operations** | | |
| **Method** | **Tickets** | **Outputs** |
| HashTable |  | HashTable |
| insert | HashTable x K key x V value | HashTable |
| searchValue | HashTable x Node<K, V> node x K key | V value |
| delete | HashTable x Node<K, V> node x K key | HashTable |

|  |
| --- |
| **Number: HashTable()** |
| Description: Creates an object of type HashTable |
| For: True |
| Post: HashTable |

|  |
| --- |
| **Number: insert()** |
| Description: Adds a value to the hash table with a key set. |
| For: True |
| Post: HashTable |

|  |
| --- |
| **Name: searchValue()** |
| Description: Returns the value of an object with the searched key |
| Pre: HashTable != Ø |
| Post: Node<K, V> goal |

|  |
| --- |
| **Name: delete()** |
| Description: Deletes a value associated with a specific key from the table |
| Pre: HashTable.size() > 0 |
| Post: HashTable |

Heap

|  |  |  |
| --- | --- | --- |
| **Abstract object: Heap = {A[0], A[1], ..., A[n]}, where A[i] is the value of the node at position i of the tree.** | | |
| A[0, n-1], where the element in A[i] has two children in A[2i+1] and A[2i+2] | | |
| **Method** | **Tickets** | **Outputs** |
| Heap() | **-** | Heap |
| maxHeapify() | Heap x arr | Heap |
| getHeap() | - | Heap arr |
| heapSort() | Heap x arr | Heap |
| getDad() | Heap x arr x int i | Node<K, V> |
| buildMaxHeap() | Heap x arr | Heap |
| getRight() | Heap x arr x int i | Node<K, V> |
| getLeft() | Heap x arr x int i | Node<K, V> |

|  |
| --- |
| **Name: Heap()** |
| **Description: Creates an object of type Heap** |
| **Pre:-** |
| **Post: Heap** |
|  |
| **Name: maxHeapify()** |
| **Description: Organizes the heap so that the current node is greater than or equal to its children** |
| **Pre: Heap** |
| **Post: Heap such that A[2i] < arr[i] and arr[2i + 1] > arr[i]** |
|  |
| **Name: heapSort()** |
| **Description: Sorts a heap in ascending order** |
| **Pre: Heap** |
| **Post: arr[i] ≤ arr[j] for all i<j** |

|  |
| --- |
| **Name: getDad()** |
| **Description: This helps us find the parent of a value** |
| **Pre: i pertains to the heap arrangement** |
| **Post: Node<K, V>** |
|  |
| **Name: getLeft()** |
| **Description: Returns the position of the left child of the current node** |
| **Pre: i pertains to the heap arrangement** |
| **Post: Node<K, V>** |
|  |
| **Name: getRight()** |
| **Description: Returns the position of the right child of the current node** |
| **Pre: i pertains to the heap arrangement** |
| **Post: Node<K, V>** |
|  |
| **Name: getHeap()** |
| **Description: Returns the array representing the heap** |
| **Pre: i <= Size(arr)** |
| **Post: Heap arr** |

# References

Singer, I. (n.d.). *Polytechnic School*. Retrieved from https://www.cartagena99.com/recursos/alumnos/apuntes/prog2-tema2-Pilas-1617.pdf

Ingeneriaudb. (n.d.). Retrieved from https://www.udb.edu.sv/udb\_files/recursos\_guias/informatica-ingenieria/programacion-con-estructuras-de-datos/2020/i/guia-8.pdf

keepcoding. (n.d.). *keepcoding.* Retrieved from https://keepcoding.io/blog/tecnicas-lifo-y-fifo-gestion-de-colas/

Lenis, A. (n.d.). Retrieved from https://blog.hubspot.es/website/interfaz-usuario#que-es

OpenAccess. (n.d.). Retrieved from https://openaccess.uoc.edu/bitstream/10609/9122/1/Intro\_TADs.pdf

pcsummary. (n.d.). *pcsummary*. Retrieved from https://www.pcresumen.com/menu-software/35-lenguajes-de-programacion/java/98-la-clase-priorityqueue-y-la-interfaz-queue#:~:text=PriorityQueue%20es%20una%20de%20las,suministra%20a%20trav%C3%A9s%20del%20constructor

sedici.unlp.edu.ar. (n.d.). Retrieved from http://sedici.unlp.edu.ar/bitstream/handle/10915/33386/Documento\_completo.pdf?sequence=3