Engineering Method

Daniel Alejandro Castro Escobar

Katerine Valens Orejuela

Icesi University

Computation and Discretes Structures

8 october 2023

**Engineering method**

***First step: Identificate the problem***

**Identification of needs:**

* **A company need to create a method to manage tasks and reminders**
* **The solution has to be quick and efficient to manage the tasks**
* **The system has to be friendly with the user**
* **The solution should allow to separate and classify the tasks in priority tasks and non priority tasks**

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

| **CLIENT** | **Discrete Structures** |
| --- | --- |
| **USER** | **Users** |
| **FUNCTIONAL REQUIREMENTS** | R1- Add tasks or reminder  R2- Modify task or reminder  R3- Delete task or reminder  R4- Display a list of tasks  R5- Management of Priority Task  R6- Management of Non-Priority Task  R7- Undo user action |
| **CONTEXT OF THE PROBLEM** | A company wishes to develop a system that allows the management of tasks and reminders. This system should allow users to coordinate their activities through functions to organize, add, manage, modify and delete their tasks. The tasks and reminders of each user should be stored with all the important information and their priority specified. The program must be able to show to the user a list of all the tasks and reminders stored, that has to be organized by the limit date or priority. The priority tasks will be managed first and the non-priority tasks will be managed according to the order of arrival. Also, the software must have a function to undo some of the actions that the user does. |
| **NON-FUNCTIONAL REQUIREMENTS** | RN1 – Implement a graphic interface  RN2 –Create a user-friendly visualization of the app |
|  |  |

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

| Name or identifier | R1 - Add tasks or reminder | | |
| --- | --- | --- | --- |
| Summary | When a user wants to add a new task, they need to provide the basic information of the task, such as: id, title, description, limit date and category of priority, and the program should be able to add the task in different structures according to their priority category. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
| key | String | - |
| Title | String | “Task Title” |
| Description | String | - |
| Limit date | Calendar | - |
| Priority | int | 1. Priority 2. Non-Priority |
| Result or postcondition | These new tasks are stored in a slot of a hash table depending on the identifier, and contains an object of type Task. These tasks contain the whole information that the user specified previously. If the task is prioritary, the system adds the task to a max heap, while if it’s not prioritary, it’s added to a queue. If the operation is successful, the program shows a message informing the user the state of his requirement. | | |
| Outputs | Output name | Data type | Selection or repetition condition |
| msg | String | “The operation is succesfull” |

| Name or identifier | R2 - Modify task or reminder | | |
| --- | --- | --- | --- |
| Summary | The software must modify the tasks that the user previously added. The user needs to select the task to modify and provide the new information of the task or reminder. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
| newTitle | String | - |
| newDescription | String | - |
| newDate | Calendar | - |
| newPriority | int | 1. Priority  2. non-priority |
| Result or postcondition | The program looks up for the task selected by the user in the hash table according to the id provided, in order to modify and replace the old information by the new one. If the operation is successful, the program will show a message informing the state of the process. | | |
| Outputs | Output name | Data type | Selection or repetition condition |
|  | msg | String | State of the operation |

| Name or identifier | R3 - Display a list of tasks | | |
| --- | --- | --- | --- |
| Summary | The software must allow the user to visualize a list of their tasks and reminders previously added. The list shows all the tasks that are added, whether they’re marked as prioritary or not. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
|  |  |  |
| Result or postcondition | The program should show to the user the list of each task stored in every slot of the hash table organized depending on the priority. | | |
| Outputs | Output name | Data type | Selection or repetition condition |
| list | string | List of every task of the user |

| Name or identifier | R4- Management of Priority Task | | |
| --- | --- | --- | --- |
| Summary | The software must use a structure of priorities for the most essential tasks. These tasks are added to a max queue according to their due date, which ensures that the task with the nearest due date is managed first. If the user marks the task as completed, the task is deleted from the hash table and from the max heap | | |
| Inputs | Input name | Data type | Selection or repetition condition |
| option | int | 1 = yes  2 = no |
| Result or postcondition | If the user selects the first option, the task is deleted from the hash table and from the max heap. In the other case, the data structures remain the same. | | |
| Outputs |  |  |  |
|  |  |  |

| Name or identifier | R5- Management of Non-Priority Task | | |
| --- | --- | --- | --- |
| Summary | The software must use a Queue for the non-priority tasks and manage them in order of arrival. The user must have the option to manage non priority tasks. When a non priority task is marked as completed, it’s erased from the respective data structures. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
| option | int | 1 = yes  2 = no |
| Result or postcondition | Non-priority tasks should be handled in the order in which they were added, so that the FIFO principle is followed. If the task is marked as completed, it’s removed from the hash table and the queue. In the other case, both data structures remain the same. | | |
| Outputs |  |  |  |
|  |  |  |

| Name or identifier | R6- Undo user action | | |
| --- | --- | --- | --- |
| Summary | The system must implement an "undo" function that allows users to revert the last action performed, whether it is adding, modifying or deleting a task. This function should use a stack (LIFO) to keep track of user actions. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
|  |  |  |
| Result or postcondition | When a user uses the undo function, the last action performed should be reverted, restoring the affected task or reminder information to its state prior to the action. | | |
| Outputs |  |  |  |
|  |  |  |

**Definition of the problem:** Create a task and reminder management system that allows users to organize and manage their to-dos, with priority options and the ability to undo actions.

***Step 2: Information Gathering***

**Definitions:**

* **Structure:** A structure is an organized way of storing data. They are very important for making different operations on some stored collections efficiently. For example, we can find some structures in computing such as : arrays, linked lists, trees, stacks, queues and many others. Each of these structures has its own characteristics, it depends on the context of the problem to use their functionalities and advantages to develop a solution.
* **Linked List**: Is a data structure used for organizing collection of nodes. Linked list store elements in the nodes that are connected to each other. Its functionality consists in a value stored in the node and pointers that allow it to create a link between two nodes making a chain starting with a head node. There are two types of list: singly linked list and doubly linked list.
* **Double list**: In a doubly linked list, each node has a pointer to both the next and the previous nodes. This makes the structure more efficient but at the same time takes more space on the memory.
* **Hash table:** A hash table is a structure for storing information efficiently. It is designed for fast data access by associating a unique key with a value. For storing data in the slot it is necessary to use a hash function to convert the key given in an index of the hash table. This allows us to retrieve the index with the value to store. Hash tables are very efficient for data retrieval because they have constant-time (O(1)) in the state of complexity and (O(n)) in the worst case.
* **Queue:** A Queue is a data structure that follows the FIFO principle (First in, first out). The first element added to the queue is the first to be removed.Queues are used when the tasks must be processed in the order of arrival. Also,it is a linear dynamic structure, this means that it has a specific order and the size can increase or decrease. The basic operations are enqueue (add an element to the tail), dequeue (remove and return an element to the tail), front, empty and full.
* **Stack:** A stack is a data structure that follows the LIFO principle (Last in first out).the last item placed on the stack is the first one to be removed. Is an object where we can insert and take elements according to the LIFO. It is limited because we can only insert and delete to the top of the stack. The basic operations are push (add at the top), pop (delete and get the top element), peek (take the top element) and empty(see if the stack is empty).
* **Heap**: Is a data structure for organizing and managing elements efficiently. They are useful for finding the maximum element or minimum element. This structure has a specific property. In a maximum heap each parent has a value larger than the value of his childs. In a minimum heap each parent node is smaller than his children.

* **Heapify:** It is an ordering algorithm that allows us to maintain the properties of the structure of a heap. This is achieved in the case of a maximum heap by exchanging nodes in subtree until the largest node is the root and in the case of a minimum heap by exchanging until the smallest node is the root, that is, until each subtree is a valid heap. This tool is fundamental for the operation of other algorithms such as heapsort.
* **HeapSort**: It is a very efficient sorting algorithm. Its function is to implement a heap to sort a structure in an order. It can also be said that it serves to order a structure in a maximum heap. It achieves this by removing and reintegrating the root node continuously until it manages to convert a simple structure to a maximum heap.

***Step 3: Search creative solutions***

In today's world we know many technologies, it is for this reason that to solve the problem posed there are thousands of possibilities. In this document we will present some of them.

**Alternative 1: Create an online schedule.** Creating an online schedule is useful because it allows the user to take notes of all the tasks and reminders. It provides many facilities such as universal access, this means that a customer can manage their tool from any device with internet connection. This alternative offers automatic synchronization, this is necessary to ensure that all the data is perfectly stored.

**Alternative 2: Develop software.** This alternative allows us to solve the problem of managing tasks and reminders, implementing different algorithms that are useful for the specified requirement of the client. The program offers the same functionalities for every customer and at the same time can be adapted to each client requirements. Developing software provides us an advantage because we have a whole universe of possible data structures to meet the needs of the user and make the experience rewarding. Also, it is more flexible than other alternatives, and gives us the opportunity to complement and improve the functionalities in the long term.

**Alternative 3: Use a virtual assistant**. This alternative provides a technological companion to the customer, this is very useful because it provides a better attention due to having an assistant make the user experience completely customized. Also improves the interaction between the system and the user. Having a virtual assistant allows the user to communicate through voice chat, an option that makes the management of tasks and reminders.

***Step 4: Preliminary designs***

Now that we have considered all the potential solutions for addressing the problem, we need to eliminate the options that are not particularly advantageous. In this regard, we will exclude Alternative 3.

**Alternative 3:** While the idea of having a virtual assistant is appealing and promises a rich user experience, it poses several challenges for the development team. In the long run, maintaining it becomes more difficult and demands increased effort from the development team. The creation and implementation of this alternative require substantial technological resources and heightened support, which can escalate project costs.

The analysis above indicates that:

**Alternative 2:**

* It is simpler to develop software using diverse structures tailored to each specific need.
* The use of various structures allows for a more personalized software.
* It can adapt more easily to changing requirements.
* The program is easier to update and maintain in the long term.
* Over time, it could potentially be more cost-effective to maintain.
* it takes a long time to develop

**Alternative 1:**

* Universal access for the customer
* The tools of the schedule allows to take notes
* It's necessary to have wifi connection for having access to the data
* The interface can be harder to use for the user
* it functionalities aren't really personalized
* It is harder to maintain

***Step 5: Selection of the best solution***

In order to choose the most optimal solution it is necessary to establish some criteria that allow us to recognize which is the best alternative to solve the problem. In this case we are going to use four criteria to compare and make a decision.

**Criteria A: User experience**

* **[2]Easy to handle and interact for the user**
* **[1]Hard to handle and interact for the user**

**Criteria B: Access Dependence**

* **[2]independent of a connection**
* **[1]dependent of a connection**

**Criteria C: Complexity in algorithmic implementation**

* **[3]Easy to implement and maintain**
* **[2]Medium to implement and maintain**
* **[1]Hard to implement and maintain**

**Criteria D: Satisfaction of the requirements**

* **[3]High Satisfaction**
* **[2]Medium Satisfaction**
* **[1]Low Satisfaction**

***EVALUATION:*** *Now we need to rate the solutions depending on the previous criteria*

| ***Solution*** | ***Criteria A*** | ***Criteria B*** | ***Criteria C*** | ***Criteria D*** | ***Total*** |
| --- | --- | --- | --- | --- | --- |
| ***Develop software*** | ***2*** | ***2*** | ***2*** | ***3*** | ***9*** |
| ***Create a online schedule*** | ***1*** | ***1*** | ***1*** | ***2*** | ***5*** |

*Since we have evaluated both solutions and implemented the criteria by way of evaluation we can conclude from the results that the most optimal solution is alternative 2, to develop a software. As we can see, the solution is easier to implement, does not require different services and allows us to provide a better user experience designing the program interactive and simple to use.*

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

**Especifiction:**

Problem: Management of task and reminders in a company

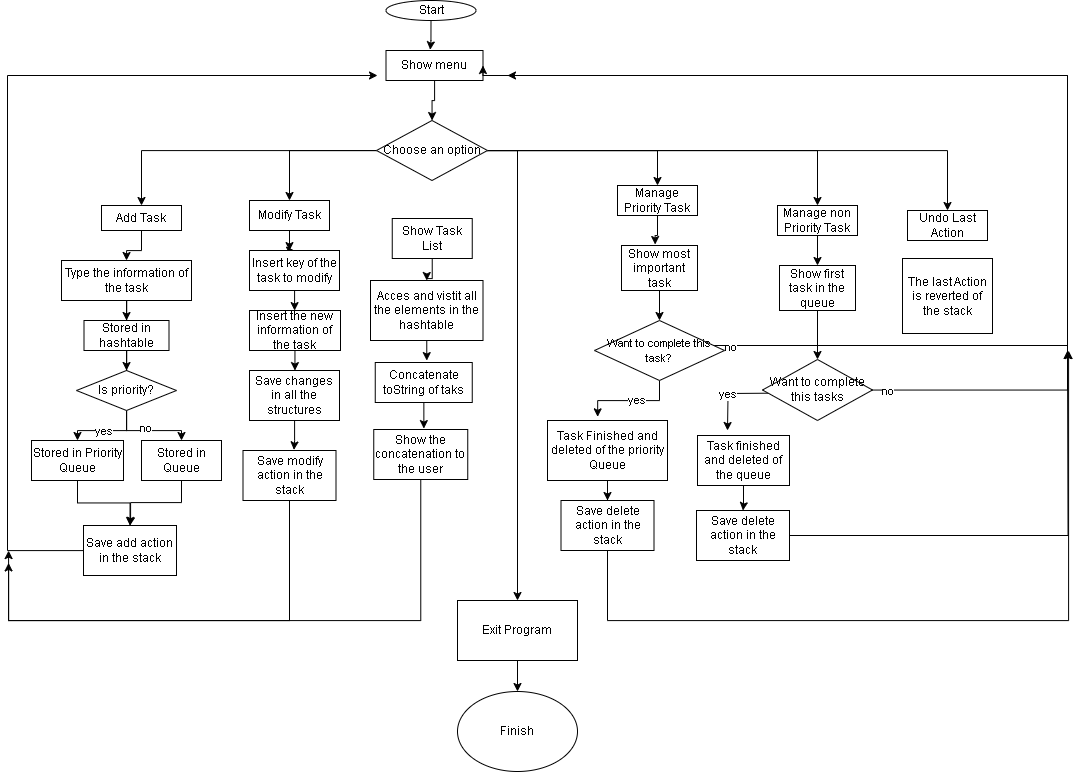
Inputs: The input depends on the functionality that the user choose, normally can be the information of a task

Outputs: The output depends on the functionality that the user choose, can be the list of tasks or messages

**Considerations:**

* **The keys that identifies the tasks and reminders should be positive to avoid problems to add them**
* **The user needs to complete the most important tasks before continue with others**

**Flow diagram:**

****

***Pseudocode: The pseudocode is going to be separated by packages***

***UTIL:***

**Class: DoubleLinkedList<T>**

- first: DoubleNode<T>

- last: DoubleNode<T>

- numberOfElements: int

Constructor DoubleLinkedList():

first = null

last = null

numberOfElements = 0

Method: isEmpty() -> bool

Return (first == null)

Method: addFirst(value: T)

newNode = Create a new DoubleNode with value

If first is null:

first = newNode

last = newNode

Else:

Set newNode's next to first

Set first's previous to newNode

Set first to newNode

Increment numberOfElements by 1

Method: addLast(value: T)

newNode = Create a new DoubleNode with value

If last is null:

first = newNode

last = newNode

Else:

Set newNode's previous to last

Set last's next to newNode

Set last to newNode

Increment numberOfElements by 1

Method: findNode(index: int) -> DoubleNode<T>

currentNode = first

If index is between 0 and numberOfElements:

localIndex = 0

While localIndex < index:

Set currentNode to currentNode's next

Increment localIndex by 1

Return currentNode

Method: delete(value: T)

currentNode = first

While currentNode is not null:

If currentNode's value equals value:

If currentNode is first:

Set first to currentNode's next

If first is not null:

Set first's previous to null

Else:

Set last to null

Else If currentNode is last:

Set last to currentNode's previous

If last is not null:

Set last's next to null

Else:

Set first to null

Else:

Set previous to currentNode's previous

Set next to currentNode's next

Set previous's next to next

Set next's previous to previous

Decrement numberOfElements by 1

Return

Set currentNode to currentNode's next

Method: deleteFirst() -> T

If first is not null:

deletedValue = first's value

If first is last:

Set first to null

Set last to null

Else:

Set first to first's next

Set first's previous to null

Decrement numberOfElements by 1

Return deletedValue

Return null

Method: deleteLast() -> T

If last is not null:

deletedValue = last's value

If first is last:

Set first to null

Set last to null

Else:

Set last to last's previous

Set last's next to null

Decrement numberOfElements by 1

Return deletedValue

Return null

Method: modifyContent(index: int, value: T)

If index is between 0 and numberOfElements:

currentNode = first

localIndex = 0

While localIndex < index:

Set currentNode to currentNode's next

Increment localIndex by 1

If currentNode is not null:

Set currentNode's value to value

Method: toString() -> string

message = ""

currentNode = first

While currentNode is not null:

Append currentNode's string representation to message

Set currentNode to currentNode's next

Return message

Method: getNumberOfElements() -> int

Return numberOfElements

Method: getFirst() -> DoubleNode<T>

Return first

Method: setFirst(first: DoubleNode<T>)

Set this.first to first

Method: getLast() -> DoubleNode<T>

Return last

Method: setLast(last: DoubleNode<T>)

Set this.last to last

**Class: DoubleNode<T>**

- value: T

- next: DoubleNode<T>

- previous: DoubleNode<T>

Constructor DoubleNode(value: T):

Set this.value to value

Set this.next to null

Set this.previous to null

Method: getValue() -> T

Return value

Method: setValue(value: T)

Set this.value to value

Method: getNext() -> DoubleNode<T>

Return next

Method: setNext(next: DoubleNode<T>)

Set this.next to next

Method: getPrevious() -> DoubleNode<T>

Return previous

Method: setPrevious(previous: DoubleNode<T>)

Set this.previous to previous

Method: toString() -> string

Return value converted to string

**Class: HashNode<K, V>**

- key: K

- value: V

- next: HashNode<K, V>

- previous: HashNode<K, V>

- status: HashNodeStatus

Constructor HashNode(key: K, value: V):

Set this.key to key

Set this.value to value

Set this.next to null

Set this.previous to null

Set this.status to HashNodeStatus.ACTIVE

Method: getKey() -> K

Return key

Method: setKey(key: K)

Set this.key to key

Method: getValue() -> V

Return value

Method: setValue(value: V)

Set this.value to value

Method: getNext() -> HashNode<K, V>

Return next

Method: setNext(next: HashNode<K, V>)

Set this.next to next

Method: getPrevious() -> HashNode<K, V>

Return previous

Method: setPrevious(prev: HashNode<K, V>)

Set this.previous to prev

Method: add(nextNode: HashNode<K, V>)

If next is null:

Set next to nextNode

Else:

Call add on next with nextNode as an argument

Method: removeElement(key: K)

If next.getKey() equals key:

Set next's status to HashNodeStatus.DELETED

Else:

If next.getNext() is not null:

Call removeElement on next.getNext() with key as an argument

Method: restoreElement(key: K) -> bool

If next.getKey() equals key:

Set next's status to HashNodeStatus.ACTIVE

Return true

Else:

If next.getNext() is not null:

Return restoreElement on next.getNext() with key as an argument

Return false

Method: getObject(key: K) -> HashNode<K, V>

If next is not null:

If next.key compareTo key equals 0:

If next's status is HashNodeStatus.ACTIVE:

Return next

Else:

Raise NonExistentKeyException with "The object with the key provided doesn't exist"

Else:

Return getObject on next with key as an argument

Else:

Raise NonExistentKeyException with "The object with the key provided doesn't exist"

Method: getStatus() -> HashNodeStatus

Return status

Method: setStatus(status: HashNodeStatus)

Set this.status to status

Method: print() -> string

msg = ""

If status is HashNodeStatus.ACTIVE:

If value is not null:

Append "Key: " + key + ", " + value converted to string to msg

If next is not null:

Append next.print() to msg

Return msg

**Class: HashTable<K, V>**

- HASH\_SIZE: int

- hashList: Array of HashNode<K, V>

- size: int

Constructor HashTable():

Set this.size to 0

Create an empty array hashList of size HASH\_SIZE

Method: hashFunction(key: K) -> int

Calculate the hash code of the key

Calculate the index using floorMod with HASH\_SIZE

Return the index

Method: isEmpty() -> bool

If size is not 0, return false; otherwise, return true

Method: getSize() -> int

Return size

Method: insertElement(key: K, value: V)

Calculate the index using hashFunction with key

Create a new HashNode<K, V> newNode with key and value

If hashList at index is not null:

Call add on hashList[index] with newNode as an argument

Else:

Set hashList at index to newNode

Increment size by 1

Method: searchElement(key: K) -> HashNode<K, V>

Calculate the index using hashFunction with key

If isEmpty is true, raise HashIsEmptyException with "The hash table is empty"

Else:

If hashList at index is null, raise NonExistentKeyException with "The object with the key provided doesn't exist"

Else If hashList at index's key compareTo key equals 0:

If hashList at index's status is HashNodeStatus.ACTIVE:

Return hashList at index

Else, raise NonExistentKeyException with "The object with the key provided doesn't exist"

Else, return getObject on hashList at index with key as an argument

Method: deleteElement(key: K)

Calculate the index using hashFunction with key

If isEmpty is true, raise HashIsEmptyException with "The hash table is empty"

Else:

If hashList at index is null, raise NonExistentKeyException with "The object with the key provided doesn't exist"

Else:

If hashList at index's next is null:

Set hashList at index to null

Decrement size by 1

Else, call removeElement on hashList at index with key as an argument

Method: restoreElement(key: K, value: V)

Calculate the index using hashFunction with key

If hashList at index is null:

Create a new HashNode<K, V> with key and value and set it at hashList[index]

Else:

If hashList at index's next is null:

If hashList at index's key equals key:

Set hashList at index's status to HashNodeStatus.ACTIVE

Else, add a new HashNode<K, V> with key and value to hashList at index

Else:

If restoreElement on hashList at index with key as an argument is false:

Add a new HashNode<K, V> with key and value to hashList at index

Method: print() -> string

msg = ""

For each index in the range 0 to the length of hashList:

If hashList at index is not null:

If hashList at index's next is null:

If hashList at index's status is HashNodeStatus.ACTIVE:

Append "Key: " + key + ", " + value converted to string to msg

Else, append the result of calling print on hashList at index to msg

Return msg

**Class: MaxHeap<T extends Comparable<T>>**

- size: int

- heap: Array of T

Constructor MaxHeap(maxsize: int)

Initialize this.heap as a new array of Comparable with maxsize

Set this.size to 0

Method: maxHeapify(i: int)

l = leftChild(i)

r = rightChild(i)

largest = i

If l is less than size and heap[l] is greater than heap[i], set largest to l

If r is less than size and heap[r] is greater than heap[largest], set largest to r

If largest is not equal to i, swap(i, largest) and call maxHeapify(largest)

Method: isEmpty() -> bool

Return whether heap[0] is null

Method: getHeap() -> Array of T

Return heap

Method: getSize() -> int

Return size

Method: setSize(heapSize: int)

Set this.size to heapSize

Method: insert(element: T)

If size is greater than or equal to the length of heap, throw HeapFullException with "The heap is full"

Set heap[size] to element

current = size

While current is greater than 0 and heap[current] is greater than heap[parent(current)], swap(current, parent(current)), and set current to parent(current)

Increment size by 1

Method: extractMax() -> T

If size is less than 1, throw PriorityQueueIsEmptyException with "The heap is empty"

max = getMax()

Set heap[0] to heap[size - 1]

Set heap[size] to null

Decrement size by 1

Call maxHeapify(0)

Return max

Method: getMax() -> T

Return heap[0]

Method: swap(i: int, j: int)

temp = heap[i]

Set heap[i] to heap[j]

Set heap[j] to temp

Method: parent(index: int) -> int

Return index / 2

Method: leftChild(i: int) -> int

Return 2 \* i + 1

Method: rightChild(i: int) -> int

Return 2 \* i + 2

Method: remove(index: int)

If index is less than 0 or index is greater than or equal to size, throw ObjectNotFoundException with "The index is not valid"

If size is 1, set heap[0] to null

Else, set heap[index] to heap[size - 1], set heap[size - 1] to null, and call maxHeapify(index)

Method: getIndexForAnObject(element: T) -> int

For each i in the range 0 to the length of heap:

If heap[i] is not null and heap[i] equals element, return i

Return -1

Method: printHeap() -> string

msg = ""

For each i in range from 0 to the length of heap:

If heap[i] is not null, append heap[i] converted to string to msg

Return msg

**Class: Queue<T>**

- list: DoubleLinkedList<T>

Constructor Queue()

Initialize this.list as a new DoubleLinkedList<T>

Method: enQueue(newElement: T)

Call addLast on list with newElement as an argument

Method: deQueue()

removedElement = Call deleteFirst on list

If removedElement is null, throw QueueIsEmptyException with "The queue is empty"

Method: front() -> DoubleNode<T>

If list is empty, throw QueueIsEmptyException with "The queue is empty"

Else, return the result of calling getFirst on list

Method: getSize() -> int

Return the result of calling getNumberOfElements on list

Method: isEmpty() -> bool

Return whether list is empty

Method: getList() -> DoubleLinkedList<T>

Return list

**Class: Stack<T>**

- list: DoubleLinkedList<T>

Constructor Stack()

Initialize this.list as a new DoubleLinkedList<T>

Method: push(newElement: T)

Call addLast on list with newElement as an argument

Method: pop()

removedElement = Call deleteLast on list

If removedElement is null, throw StackIsEmptyException with "The stack is empty"

Method: top() -> DoubleNode<T>

If list is empty, throw StackIsEmptyException with "The stack is empty"

Else, return the result of calling getLast on list

Method: getSize() -> int

Return the result of calling getNumberOfElements on list

Method: isEmpty() -> bool

Return whether list is empty

***PACKAGE MODEL***

**Class: Action**

- actionType: ActionType

- task: Task

- originalTask: Task

Constructor Action(actionType: ActionType, task: Task)

Set this.actionType to actionType

Set this.task to task

Constructor Action(actionType: ActionType, task: Task, originalTask: Task)

Set this.actionType to actionType

Set this.task to task

Set this.originalTask to originalTask

Method: getActionType() -> ActionType

Return actionType

Method: setActionType(actionType: ActionType)

Set this.actionType to actionType

Method: getTask() -> Task

Return task

Method: setTask(task: Task)

Set this.task to task

Method: getOriginalTask() -> Task

Return originalTask

Method: setOriginalTask(originalTask: Task)

Set this.originalTask to originalTask

**Class: Controller**

- hashTableTask: HashTable<Integer, Task>

- actions: Stack<Action>

- queueTask: Queue<Task>

- heapTask: MaxHeap<Task>

Constructor Controller()

Initialize hashTableTask as a new HashTable<Integer, Task>

Initialize actions as a new Stack<Action>

Initialize queueTask as a new Queue<Task>

Initialize heapTask as a new MaxHeap<Task> with a max size of 200

Method: addTask(name: String, description: String, strLimitDate: String, priorityLevel: int, key: int) throws HeapFullException

Split strLimitDate by "/"

Convert parts to year, month, and day

Create a new GregorianCalendar instance with year, month-1, and day

Set priority based on priorityLevel

Create a new Task with name, description, key, limitDate, and priority

If priority is PRIORITY

Insert newTask into heapTask

Else if priority is NON\_PRIORITY

Enqueue newTask into queueTask

Insert newTask into hashTableTask

Create a new Action of type ADD with newTask and push it onto actions

Method: modifyTask(key: int, newName: String, newDescription: String, newStrLimitDate: String, newPriorityLevel: int) throws HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, HeapFullException, CloneNotSupportedException

Retrieve task from hashTableTask using key

Clone task to originalTask

Split newStrLimitDate by "/"

Convert parts to year, month, and day

Create a new GregorianCalendar instance with year, month-1, and day

Set newPriority based on newPriorityLevel

Update task with newName, newDescription, newLimitDate, and newPriority

If currentPriority is not equal to newPriority

If currentPriority is PRIORITY

Find and remove task from heapTask

Enqueue task into queueTask

Else if currentPriority is NON\_PRIORITY

Delete task from queueTask

Insert task into heapTask

Create a new Action of type MODIFY with task and originalTask, and push it onto actions

Method: showAllTasks() -> String

Return the result of calling print on hashTableTask

Method: showPrioritaryTasks() -> String

Return the result of calling printHeap on heapTask

Method: showNonPrioritaryTasks() -> String

Return the result of calling toString on queueTask.getList()

Method: showFirstPrioritaryTask() -> String throws PriorityQueueIsEmptyException

Return the result of calling toString on getMax of heapTask

Method: showFirstNonPrioritaryTask() -> String throws QueueIsEmptyException

Return the result of calling toString on front of queueTask

Method: managePriorityTask() throws PriorityQueueIsEmptyException, HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, QueueIsEmptyException

Extract the maximum task from heapTask

Get the key of the currentTask

Retrieve the hashNode from hashTableTask using the key

Set the status of the hashNode to DELETED

Create a new Action of type COMPLETE with the currentTask and push it onto actions

Method: manageNonPriorityTask() throws QueueIsEmptyException, HashIsEmptyException, NonExistentKeyException

Get the front task from queueTask

Get the key of the currentTask

Retrieve the hashNode from hashTableTask using the key

Set the status of the hashNode to DELETED

Dequeue the currentTask from queueTask

Create a new Action of type COMPLETE with the currentTask and push it onto actions

Method: revertLastAction() throws StackIsEmptyException, HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, HeapFullException

Get the last action from actions

Get the task from the last action

If the last action is of type ADD

If task is PRIORITY

Find and remove task from heapTask

Else if task is NON\_PRIORITY

Delete task from queueTask

Delete task from hashTableTask

Else if the last action is of type MODIFY

Get the originalTask from the last action

Set the name, description, limitDate, and priority of task to match originalTask

Else (the last action is of type COMPLETE)

If task is PRIORITY

Insert task into heapTask

Else if task is NON\_PRIORITY

Enqueue task into queueTask

Restore task in hashTableTask using key

Method: getHeapSize() -> int

Return the size of heapTask

**Class: Task**

- name: String

- description: String

- key: int

- limitDate: Calendar

- priorityLevel: PriorityLevel

Constructor Task(name: String, description: String, key: int, limitDate: Calendar, priorityLevel: PriorityLevel)

Set name to the provided name

Set description to the provided description

Set key to the provided key

Set limitDate to the provided limitDate

Set priorityLevel to the provided priorityLevel

Method: getName() -> String

Return the name

Method: setName(name: String)

Set the name to the provided name

Method: getDescription() -> String

Return the description

Method: setDescription(description: String)

Set the description to the provided description

Method: getLimitDate() -> Calendar

Return the limitDate

Method: setLimitDate(limitDate: Calendar)

Set the limitDate to the provided limitDate

Method: getLimitDateString() -> String

Create a SimpleDateFormat instance with the format "yyyy/MM/dd"

Format the limitDate using the SimpleDateFormat

Return the formatted date as a string

Method: getPriorityLevel() -> PriorityLevel

Return the priorityLevel

Method: setPriorityLevel(priorityLevel: PriorityLevel)

Set the priorityLevel to the provided priorityLevel

Method: compareTo(otherTask: Task) -> int

Compare the limitDate of this task with the limitDate of otherTask

Return the result of the comparison (negative if this task's date is earlier, positive if it's later)

Method: toString() -> String

Return a string with the task's name, description, formatted limitDate, and priorityLevel

Method: getKey() -> int

Return the key

Method: clone() -> Object

Call the superclass's clone method and return the result

***Finally we have the package ui where is the main class***

***Class: Main***

- sc: Scanner

- controller: Controller

Constructor Main()

Initialize sc with a new Scanner reading from System.in

Initialize controller with a new Controller

Method: main(String[] args)

Create a new Main instance (main)

Initialize option with -1

Loop (while option is not 0)

Call showMenuAndGetOption() to get userOption

Call answerOption(userOption) to handle the user's choice

End Loop

Method: answerOption(int userOption)

Switch on userOption

Case 0:

Print "¡Goodbye!"

Break

Case 1:

Call addTask()

Break

Case 2:

Call modifyTask()

Break

Case 3:

Call showAllTasks()

Break

Case 4:

Call managePriorityTask()

Break

Case 5:

Call manageNonPriorityTask()

Break

Case 6:

Call revertLastAction()

Break

Case 9:

Call testCases()

Break

End Switch

Method: showMenuAndGetOption() -> int

Print the main menu

Read an integer from the user and assign it to input

Return input

Method: addTask()

Print prompts and read user input for task details (title, description, date, state, key)

Try

Call controller.addTask(title, description, date, state, key)

Catch HeapFullException e

Print e.getMessage()

Method: modifyTask()

Read user input for the key of the task to modify

Read user input for the new task details (title, description, date, state)

Try

Call controller.modifyTask(key, title, description, date, state)

Catch HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, HeapFullException, CloneNotSupportedException e

Print e.getMessage()

Print "The task was added successfully"

Method: showAllTasks()

Call controller.showAllTasks()

Print the result

Method: managePriorityTask()

Try

If controller.showPrioritaryTasks() is not empty

If controller.getHeapSize() is not 0

Print "This is the priority task with the nearest due date"

Print controller.showFirstPrioritaryTask()

Print "Would you like to mark this task as completed?"

Read option

If option is 1

Try

Call controller.managePriorityTask()

Catch ObjectNotFoundException, QueueIsEmptyException e

Print e.getMessage()

End Try

End If

Else

Print "There aren't any pending prioritary tasks to manage"

End If

Else

Print "There aren't any pending prioritary tasks to manage"

End If

Catch PriorityQueueIsEmptyException, HashIsEmptyException, NonExistentKeyException e

Print e.getMessage()

Print "The task was successfully completed"

Method: manageNonPriorityTask()

Try

If controller.showNonPrioritaryTasks() is not empty

Print "This is the first non-priority task registered"

Print controller.showFirstNonPrioritaryTask()

Print "Would you like to mark this task as completed?"

Read option

If option is 1

Try

Call controller.manageNonPriorityTask()

Catch HashIsEmptyException, NonExistentKeyException e

Print e.getMessage()

End Try

End If

Else

Print "There aren't any pending non-priority tasks to manage"

End If

Catch QueueIsEmptyException e

Print e.getMessage()

Method: revertLastAction()

Try

Call controller.revertLastAction()

Catch StackIsEmptyException, HashIsEmptyException, NonExistentKeyException, ObjectNotFoundException, HeapFullException e

Print e.getMessage()

Method: testCases()

Try

Call controller.addTask("name", "null", "2022/11/01", 1, 12)

Call controller.addTask("name2", "null", "2022/11/02", 1, 1)

Call controller.addTask("name3", "null", "2022/11/03", 2, 15)

Call controller.addTask("name4", "null", "2022/11/04", 2, 18)

Catch HeapFullException e

Print e.getMessage()

***Step 7: Implementation***

***List of functionalities implemented:***

* **Add task**
* **Modify task**
* **Show list of tasks**
* **Manage priority tasks**
* **Manage non priority tasks**
* **Undo last action**

***Implementation:***

**We used java to develop the system and all its functionalities, the code is in the github repository:**

[***https://github.com/tobivalens/first-task/tree/main***](https://github.com/tobivalens/first-task/tree/main)