**ENGINEERING METHOD**

**Problematic context**

The development of a task and reminder management system that allows users to add, organize and manage tasks and reminders is required. The management system must have the functionality to store, manage tasks and reminders as well as user actions.

**Step 1: Problem Identification**

Identification of needs and symptoms:

* Users of the task and reminder management system need to add, organize, and manage their pending tasks and reminders.
* A condition to be resolved is that storing tasks and reminders should be done through hash tables.
* Users should be able to view a list of all tasks and reminders, sorted by deadline or priority.
* Sorting functionality should use heapsort.
* The solution should have two categories for user tasks: priority for performing tasks according to their level of importance, allowing important tasks to be done first, and non-priority tasks to manage tasks that have not been prioritized.
* Users can undo their actions in the system, and this should be resolved using stacks.

*(Requirements specification is in another document called "requirementsT1")*

**Step 2: Information Gathering**

Search for definitions of terms to be implemented in the problem:

*Temporal Complexity*

Temporal complexity is the number of operations an algorithm performs to complete its task (considering that each operation takes the same amount of time). The algorithm that accomplishes the task with the fewest operations is considered the most efficient in terms of temporal complexity. However, spatial and temporal complexity are affected by factors such as the operating system and hardware, but we will not include them in the discussion.

*Spatial Complexity*

Memory used by a program for its execution, indicating the amount of space required to run the algorithm; that is, the space in memory occupied by all variables specific to the algorithm.

*Stacks*

A stack is an ordered collection of elements in which data is inserted or removed from the same end called the "top" of the stack.

*Queues*

A queue (also called a line) is an abstract data type characterized by being a sequence of elements in which the push insertion operation is performed at one end and the pull extraction operation at the other. It is also called a FIFO (First In First Out) structure because the first element to enter will also be the first to exit.

*Hash tables*

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

*Big O*

Big O notation is one of the most fundamental tools for computer scientists to analyze the cost of an algorithm. It is a good practice for software engineers to understand in-depth as well.

*Parameterization*

Parameterization is the process of taking values or objects defined within a function or a method, and making them parameters to that function or method, in order to generalize the code. This process is also known as the “extract parameter” refactoring.

*Unit testing*

Unit testing is an essential instrument in the toolbox of any serious software developer. However, it can sometimes be quite difficult to know how to write unit tests for a particular piece of code. Having difficulty testing their own or someone else’s code, developers often think that their struggles are caused by a lack of some fundamental testing knowledge or secret unit testing techniques.

*Arrays*

An array is an arrangement of numbers, pictures or objects formatted into rows and columns according to their type. In coding and programming, an array is a collection of items, or data, stored in contiguous memory locations, also known as database systems. The purpose of an array is to store multiple pieces of data of the same type together. You can use an array to demonstrate a mathematical property known as the commutative property of multiplication, which illustrates that you can change the order of the factors or elements, and the product of the elements remains the same.

In computer programming, an array can help you locate and identify where you stored each piece of data, or element, by adding an offset to each value. An offset is a number that represents the difference between the two indexes. Similar to an index in a book, an index in computer programming contains a record of entries with the names of the data items and their locations. Thus, you can identify each element and its location by referring to the index within the array.

*Generics*

Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters.

*Arraylist*

The ArrayList class is a resizable array, which can be found in the java.util package. The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one).

*Heapsort*

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

**Step 3: Creative Solutions Search**

Divide the problem into subsystems, considering different ways to approach each subsystem, such as the types of structures and methods needed.

We will tackle the problem by dividing it into three main parts: storing tasks, prioritizing tasks, and managing user actions. For each subsystem, we consider various data structures and methods.

**3.1. Storing Tasks:**

* Hash Tables
* ArrayList
* Array
* Linked List
* Queue (FIFO)
* Priority Queue

**3.2 Prioritizing Task Management:**

* Queue (FIFO)
* Stacks (LIFO)
* Hash Tables
* FIFO Stacks

**3.3 User Action Management:**

* LIFO Stacks
* FIFO Stacks
* Queues
* Linked Lists
* Trees

For the storage part, it is understood that we need a storage structure, which can be arrays, ArrayLists, linked lists, and hash tables. For managing prioritized tasks, we could use ArrayLists.

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

Here, we will divide the options into sections and, based on this, eliminate less feasible ideas.

**4.1. Storing Tasks:**

* Hash Tables: Efficient access, but may consume more memory.
* ArrayList: Quick access by index, less efficient for mid-position insertions and deletions.
* Array: Quick access, fixed size.
* Linked List: Efficient for insertions and deletions at any position.
* Queue (FIFO): Useful for maintaining strict order.
* Priority Queue: Efficient for managing prioritized tasks.

**4.2. Prioritizing Task Management:**

* Queue (FIFO): Useful for managing prioritized tasks based on FIFO.
* Stacks (LIFO): Useful for managing prioritized tasks based on LIFO.
* Hash Tables: Efficient access to prioritized tasks based on keys.
* Priority Queue: Efficient for managing prioritized tasks.

**4.3.** **User Action Management:**

* Stacks (LIFO): Useful for tracking user actions in reverse order.
* Queues (FIFO): Useful for recording user actions in the exact order they occurred.
* Linked Lists: Useful for maintaining an ordered record of user actions.
* Trees: Useful for a more complex data structure to represent the sequence of user actions.

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

This step will be performed to find the best solution to implement, creating a numerical evaluation system based on the following criteria:

1. Efficiency
2. Usability
3. Maintainability
4. Scalability

Each point will be rated from 1 to 5, with 1 being very poor and 5 being excellent. At the end of each case, a sum will be made, and the higher the value, the more convenient it will be to use.

**5.1 Storing Tasks:**

Hash Tables:

* Efficiency: Provides quick and efficient access to tasks. (5)
* Usability: May require unique keys and consumes more memory. (4)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable for a large number of tasks. (5)

Total: 18

ArrayList:

* Efficiency: Quick access by index, less efficient for insertions and deletions. (3)
* Usability: Easy to use and understand. (4)
* Maintainability: Reasonable maintenan. (4)
* Scalability: Scalable, but not the best choice for large amounts of ever-changing data. (3)

Total: 14

Array:

* Efficiency: Quick access by index, less efficient for insertions and deletions. (3)
* Usability: Easy to use and understand, but not as user-friendly as ArrayList. (2)
* Maintainability: Reasonably maintainable, but may require more effort compared to ArrayList. (3)
* Scalability: Scalable, but not the best choice for large amounts of ever-changing data. (3)

Total: 11

Linked lists:

* Efficiency: Efficient for insertions and deletions in any position. (4)
* Usability: A little more complex to use than ArrayList. (3)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Suitable for a variable number of tasks. (4)

Total: 15

Queue (FIFO):

* Efficiency: Useful to maintain a strict order (FIFO). (4)
* Usability: Useful for tasks that must be processed in order of arrival. (4)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable to manage queued tasks. (4)

Total: 16

Priority Queue:

* Efficiency: Efficient in managing priority tasks. (4)
* Usability: Excellent for prioritized tasks. (5)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable and suitable for prioritizing tasks. (4)

Total: 17

**5.2. Task Prioritization Management:**

Queue (FIFO):

* Efficiency: Useful for managing prioritized tasks based on FIFO. (5)
* Usability: Easy to use. (4)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable for managing prioritized tasks. (4)

Total: 17

Stacks (LIFO):

* Efficiency: Useful for managing prioritized tasks based on LIFO. (4)
* Usability: Easy to use. (4)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable for managing prioritized tasks. (4)

Total: 16

Hash Tables:

* Efficiency: Provides fast access to prioritized tasks based on keys. (4)
* Usability: May require unique keys. (3)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable for managing prioritized tasks. (4)

Total: 15

Priority Queue:

* Efficiency: Efficient for managing prioritized tasks. (4)
* Usability: Excellent for tasks with priorities. (5)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable and suitable for prioritizing tasks. (4)

Total: 17

**5.3. User Action Management:**

Stacks (LIFO):

* Efficiency: Useful for tracking user actions in reverse order (LIFO). (4)
* Usability: Easy to use. (4)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable for recording user actions. (4)

Total: 16

Queues (FIFO):

* Efficiency: Useful for recording user actions in the exact order they occurred (FIFO). (4)
* Usability: Easy to use. (4)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable for recording user actions. (4)

Total: 16

Linked Lists:

* Efficiency: Efficient for maintaining an ordered record of user actions. (4)
* Reasonably easy to use. (3)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Suitable for recording user actions. (4)

Total: 15

Trees:

* Efficiency: Potentially efficient for representing the sequence of user actions. (3)
* Usability: More complex to implement and use compared to other options. (2)
* Maintainability: Reasonable maintenance. (4)
* Scalability: Scalable for more complex action structures. (3)

Total: 13

**Selection**

For storing taskswe will use hash tables because with this, we can use keys to unlock a value. In other words, we can use a unique identifier as a key to retrieve information about the task/reminder.

For task prioritization management we will use queues and priority queues. Queues will organize tasks based on their level of importance, and queues will manage non-priority tasks according to the first-in, first-out (FIFO) principle.

For the user action managementwe will use stacks and possibly queues. Using a stack will enable us to use the Last-In, First-Out (LIFO) principle to track actions that have already been taken, as well as to implement the undo function.

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

Problem Specification (in terms of input/output):

Problem: Task and reminder management system.

Inputs:Task/reminder identifier, title, description, deadline and priority (Priority or Non-priority) of the task.

Output: According to the action, a message indicating the result of each operating or a list of tasks and reminders sorted by order of arrival and priority.

Considerations:

About the system in general:

* The system must allow users to add, modify, and delete tasks and reminders.
* Tasks and reminders are stored in a hash table, where the key is a unique identifier, and the value contains information about the task/reminder.
* Two categories are used for tasks: "Priority" and "Non-priority."
* Priority tasks are managed using a priority queue.
* Non-priority tasks are managed based on their arrival order (FIFO).
* An "undo" functionality is implemented using a stack (LIFO) to track user actions.

About the "undo" functionality (how to implement it):

Create action stack: Create a stack to track user actions. Each time the user performs an action (add, modify, or delete a task), the action is recorded in the stack. Each stack entry contains information about the action performed and details of the affected task.

Record actions: Every time the user performs an action, record the action in the stack. For example, if the user adds a new task, record the action as "Add task" along with the task details. If the user modifies a task, record "Modify task" and include the details before and after the modification.

Undo Method: Implement a method that allows undoing the last action performed by the user. This method pops the last action from the stack and reverses the corresponding action based on the information stored in the stack.

Use of the Undo Method: In the user interface, provide users with the option to undo the last action performed. When the user selects the "Undo" option, call the undo() method, which will reverse the last action performed.

*(Class diagram is in another document called "Task reminder system")*

Pseudocode of the main system functionalities:

To add an element:

String addElement(id, tittle, description, dueDate, isTask, importance):

Try:

validateDueDate(dueDate);

TaskReminder element = new Element(id, title, isTask, dueDate, importance);

taskReminderTable.insert(id, element);

if(isTask && importance != 0):

priorityTasks.insert(element);

return “Priority task added.”;

else if(isTask):

nonPriorityTasks.enqueue(id, reminder);

return “Non-priority task added.”;

else:

return “Reminder added.”;

actions.push(id, new Action(“Add element”, element));

Catch (HeapSizeException, DuplicatedObjectException, InvalidDateException)

return getExceptionMessage();

To edit an element:

String editElement(id, title, description, dueDate, isTask, importance):

Try:

validateDueDate(dueDate);

TaskReminder element = taskReminderTable.search(id).getValue();

if(element != NIL):

element.setTitle(title);

element.setDescription(description);

element.setDueDate(dueDate);

int importanceCopy = element.getImportance();

element.setImportance(importance);

if (isTask && importance != 0):

if (importanceCopy < importance):

priorityTasks.maxHeapify(0);

else:

int i = priorityTasks.searchTaskIndex(element);

priorityTasks.increaseKey(i, element);

return “Priority task edited.”;

else if (isTask):

Node queueNode = nonPriorityTasks.search(id);

queueNode.getValue().setTitle(title);

queueNode.getValue().setDescription(description);

queueNode.getValue().element.setDueDate(dueDate);

return “Non-priority task edited.”;

else:

return “Reminder editet.”;

actions.push(id, new Action(“edit element”, element));

else:

return “Element with the entered ID does not exist.”;

Catch (InvalidDateException)

return GetExceptionMessage();

To delete an element:

String deleteElement(id):

TaskReminder element = taskReminderTable.search(id).getValue();

if (element != NIL):

String targetH = taskReminderTable.search(id).getPrevious().getId();

TaskReminderTable.delete(id);

if (element.isTask() && element.getImportance() != 0):

deletePriorityTask(element);

actions.push(id, new Action(“Delete element”, element, targetH));

return “Priority task deleted.”;

else if (element.isTask()):

Node queueNode = nonPriorityTasks.search(id);

String targetQ = queueNode.getPrevious().getValue().getId();

deleteNonPriorityTask(element);

actions.push(id, new Action(“Delete element”, element, targetH, targetQ))

return “Non-priority task deleted.”;

else:

actions.push(id, new Action(“Delete element”, element, targetH));

return “Reminder deleted.”;

else:

return “Element with the entered ID does not exist.”;

To undo an action:

void undoAction():

if (!actions.isEmpty()):

Action action = actions.pop();

String actionType = action.getType();

TaskReminder record = action.getRecord();

String id = record.getId();

if (actionType.equals("Add element")):

deleteElement(id);

else:

int importance = record.getImportance();

if (actionType.equals("Edit element")):

String title = record.getTitle();

String description = record.getDescription();

Date dueDate = record.getDate();

boolean isTask = record.isTask();

editElement(id, title, description, dueDate, isTask, importance);

else;

String targetH = action.getTargetH();

returnElementToHash(targetH, record);

if (importance != 0):

priorityTasks.insert(record);

else:

String targetQ = action.getTargetQ();

returnElementToQueue(targetQ, record);

*Note: These pseudocodes are a simple version and are for understanding the flow. In the final implementation they need to be more complex.*

**Step 7. Implementation of the Design**

Task Lists to Implement:

1. Store Tasks and Reminders in a Hash Table
2. User Interface
3. Priority Management
4. Implementation of "Undo" with a Stack

**Sources**

* <https://www.freecodecamp.org/espanol/news/introduccion-a-la-complejidad-temporal-de-los-algoritmos/>
* <http://aniei.org.mx/paginas/uam/CursoAA/curso_aa_02.html#:~:text=Complejidad%20Espacial%3A%20Memoria%20que%20utiliza,las%20variables%20propias%20al%20algoritmo>.
* <https://ccc.inaoep.mx/ingreso/programacion/corto2015/Curso-PROPE-PyED-5-Pilas-Colas.pdf>
* <https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica)>
* <https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm>
* <https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/>
* <https://www.toptal.com/python/python-parameterized-design-patterns#:~:text=Parameterization%20is%20the%20process%20of,the%20%E2%80%9Cextract%20parameter%E2%80%9D%20refactoring>.
* <https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters>
* <https://www.w3schools.com/java/java_arraylist.asp#:~:text=The%20ArrayList%20class%20is%20a,to%20create%20a%20new%20one>
* <https://www.geeksforgeeks.org/heap-sort/>