Step 1. Problem Identification (Provide Context, State Needs and Symptoms, Define the Problem)

TaskMaster, the company, seeks our collaboration in developing an application designed to assist individuals in managing their daily tasks and activities. This initiative arises from the observation that many people experience difficulties in prioritizing their daily responsibilities, which can lead to neglecting important events such as appointments, business meetings, or birthdays.

In this context, effective activity management relies on two fundamental aspects: the chronological order of events and their level of priority. Furthermore, it is crucial to consider the user profiles that the application is targeting since the needs of a university student may differ significantly from those of a professional who must manage their daily work tasks.

To address these needs, we have identified a series of essential features and functionalities that should be part of the application:

**Storage of Tasks and Reminders:** The ability to record and store tasks and reminders with detailed information, including title, description, deadline, and priority level. To achieve this, we are considering the use of a hash table that allows for fast data retrieval.

**Intuitive User Interface:** A user-friendly interface that enables users to easily add, modify, or delete tasks and reminders. This interface should also provide an organized view of tasks, allowing sorting by deadline or priority, and it will include sorting functionality using the heapsort algorithm.

**Priority Management:** The ability to categorize tasks as "Priority" and "Non-priority" to help users focus on the most important tasks. This will be implemented using a priority queue.

**Focus on Non-priority Tasks:** Inclusion of a category for non-priority tasks, managed on a first-come, first-served basis (FIFO), to prevent the accumulation of lower-urgency responsibilities.

**Undo Action Method**: Implementation of a method that allows users to undo the last action performed in the system. This will be achieved through the use of a stack (LIFO) that records user actions and facilitates the reversal of unwanted actions.

This set of features is essential to address the initial needs of the application and lay the groundwork for effective task and reminder management. As we progress in development, we will consider how to adapt the interface for different user profiles, ensuring a personalized and efficient experience.

**Needs and Symptoms Identified:**

**Task Accumulation:** Individuals often face a constant accumulation of tasks and reminders as they progress through their day. This can lead to a sense of disorder and the possibility of overlooking important tasks.

**Difficult Prioritization:** Distinguishing between priority and non-priority tasks can be challenging, leading to inefficient time management.

**Lack of Tracking:** People may forget to complete important tasks or attend scheduled events due to the absence of an effective reminder system.

**Management Errors:** Users may make mistakes when adding, modifying, or deleting tasks, which can result in data loss or confusion in task management.

**Need for Flexibility:** Users require the ability to undo actions in case of errors or changes in task or reminder management.

**Problem Definition:**

The fundamental problem lies in the absence of an effective task and reminder management tool that meets users' needs for organization and prioritization. Current solutions lack an intuitive interface, an efficient way to prioritize tasks and reminders, and the ability to undo actions in case of errors. Additionally, they do not provide a systematic way to store, organize, and retrieve tasks and reminders. Therefore, there is a need for the development of a task and reminder management system that addresses these challenges and provides a comprehensive solution for organizing and managing tasks and reminders efficiently.

**Functional Requirements:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Identifier and name** | *[RF1-* ***Recording of Tasks and Reminders*** *]* | | | |
| **Summary** | * Allows registering tasks and reminders in the system. * Each task must have at least a title and a description. * Users can set a deadline date and time for tasks and reminders. * It should be possible to assign a priority level (high, medium, low) to each task. | | | |
| **Entries** | **Name** | **Type** | | **Condition** |
| Info, key | String, Int | | *"Positive numbers can be used in the password, and both uppercase and lowercase letters are allowed."* |
| **Result or Postcondition** | "A confirmation message after the information has been registered." | | | |
| **Exit** | **Name** | | **Type** | **Format** |
| msg | | String | *"Warning Message"* |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * **Identifier and name** | *[RF2-* **Manage Activities by Priorities***]* | | | |
| **Summary** | * Tasks should be categorized by the user as "Priority" or "Non-priority" based on their priority level. * Users can change the categorization of a task if they wish to do so. | | | |
| **Entries** | **Name** | **Type** | | **Conditions** |
| key, type | Int | | Positive numbers for both, and they must be different from 0. |
| **Result o Postcondition** | The status of an activity is changed, and the activity is recorded. | | | |
| **Exit** | **Name** | | **Type** | **Format** |
| msg | | string | “Confirmation Message" |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * **Identifier and name** | *[RF3-* **"Record Non-Priority Tasks"***]* | | | |
| **Summary** | * There should be a special category for non-priority tasks. * Non-priority tasks are managed based on their arrival order (FIFO). | | | |
| **Entries** | **Name** | **Type** | | **Condition** |
| Key, type | Int | | Positives values |
| **Result o Postcondition** | "Depending on the type, a distinction will be made between a priority task, and it will be managed based on its arrival order or date." | | | |
| **Exit** | **Name** | | **Type** | **Format** |
| msg | | String | "Confirmation Message" |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * **Identifier and name** | *[RF4-* **"Sorting and Search"***]* | | | |
| **Summary** | * Users can sort the list of tasks and reminders by deadline, priority, or title. * There must be a search function that allows users to find specific tasks using keywords. | | | |
| **Entries** | **Name** | **Type** | | **Conditions** |
| key | String | | Ensure the existence of the object |
| **Result o Postcondition** | The person will be shown the activity they were looking for and can arrange it by date, priority, or title. | | | |
| **Exit** | **Name** | | **Type** | **Format** |
| Msg | | String | “"Confirmation Message" |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * **- Identifier and name.** | *[RF5 -* **"Undo Actions"***]* | | | |
| **Sumary** | * Users can undo the last action performed in the system, such as adding, modifying, or deleting a task. * The undo function should be easily accessible from the interface. | | | |
| **Entries** | **Name** | **Type** | | **Conditions** |
| Key, name, info | String | | Ensure that the keys do not repeat. |
| **Result or Postcondition** | "A confirmation message will be displayed." | | | |
| **Exit** | **Name** | | **Type** | **Format** |
| msg | | String | "Changes Made" or "Task Added" |

**Non-Functional Requirements:**

**Notifications:** Users should receive notifications or alerts via email, text messages, or in-app push notifications to remind them of pending tasks before their deadlines.

**Performance:** The application must be fast and efficient in retrieving and displaying tasks and reminders, even when dealing with extensive lists.

**Availability:** The application must be available to users at all times, ensuring high uptime and minimal downtime.

**Compatibility:** The application should be compatible with a variety of web browsers and popular mobile devices to reach a broad user base.

**Intuitive User Interface:**

**-** The user interface should be easy to use and accessible on both mobile and web devices.

- Users should be able to view a well-organized and sorted list of tasks and reminders.

- The interface should allow users to easily modify, mark as complete, or delete tasks and reminders.

**Scalability:** The application must be scalable to handle an increase in the number of users and data without compromising performance.

**Documentation:** Clear documentation for end-users on how to use the application should be provided, including user guides and instructions.

**Decoupling or Ease of Maintenance for Future Changes:** It should be easy for the development team to perform updates and enhancements in the application.

Step 2. Information Gathering (Collect Information about the Data Structures to Be Used)

**Hash Table:** A hash table is a data structure that associates keys (unique identifiers) with values (related information). It allows for quick data access through a hash function that transforms the key into an address in the table where the corresponding value is stored. When implementing the hash table, the initial size and the table's expansion capacity should be considered. In this case, a variable-sized table that automatically expands when a certain load is reached is planned. Regarding hash table collisions, two approaches are considered: chaining or open addressing.

**Heapsort Algorithm:** Heapsort is a sorting algorithm that uses a data structure called a heap to organize elements and then extract them in ascending or descending order. It is efficient and suitable for sorting large datasets. In this case, efficient implementation of the heapsort algorithm is planned, ensuring that the operation of extracting elements from the heap is performed correctly to obtain tasks sorted by deadline or priority.

**Priority Queue:** Use a minimum priority queue for prioritized tasks and a maximum priority queue for non-prioritized tasks. A priority queue is a data structure that stores elements with an associated priority value. It allows for inserting elements based on their priority and efficiently retrieving the element with the highest or lowest priority.

**FIFO (First-In, First-Out):** FIFO is an organizational principle where the first element to enter a structure or queue is the first to be removed or processed. It is commonly used to manage tasks in the order they are received. Tasks should be enqueued in the order they arrive and dequeued similarly.

**LIFO (Last-In, First-Out):** When users wish to undo an action, reverse the last action performed using the Last-In, First-Out (LIFO) stack of actions. Ensure that actions are undone fully and accurately. LIFO is another organizational principle where the last element to enter a structure or stack is the first to be removed or processed. It is planned to be used to implement the "undo" functionality.

**User Interface (UI):** The user interface refers to the part of an application or system that allows interaction between the user and the machine. It includes visual and functional elements that facilitate communication and user actions. For the planned operation of the application, it is expected that users will be able to understand the system due to the intuitiveness of the module.

**Push Notification:** A push notification is a message sent instantly through the application to a mobile device or computer. It is used to inform users about relevant application events or updates. Users might have the option to configure their notification preferences, such as the frequency and type of notifications they wish to receive based on dates and priorities.

Step 3. Solution Search (For each data structure, a solution is proposed for the general problem)

**Hash Table:**

* Hash Table Implementation: Use a hash table to store tasks and reminders. Each entry in the hash table could contain information such as title, description, deadline, priority, etc.
* Hashing Function: Design an efficient hashing function that can convert the unique identifier of each task into an address in the hash table.
* Hash Collision: Implement a strategy to handle hash collisions, such as resolving them through chaining or rehashing.

**Heapsort Algorithm:**

* Application of Heapsort: Utilize the heapsort algorithm to sort tasks by deadline or priority. Build a heap of tasks and extract elements in the desired order.
* Heap Maintenance: Ensure that the heap is properly maintained as users add, modify, or delete tasks.

**Priority Queue:**

* Task Categorization: Implement a priority queue to categorize tasks as "Priority" or "Non-priority" based on their level of importance.
* Management of Priority Changes: Allow users to change the categorization of a task if its priority level changes.

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

* Non-priority Tasks: Use the FIFO principle to manage non-priority tasks. As users add non-priority tasks, place them in a simple queue and process tasks in the order they arrive.

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

* Undo Method: Implement a Last-In, First-Out (LIFO) stack to track user actions. Each time a user adds, modifies, or deletes a task, record the action on the stack.

**User Interface (UI):**

* Intuitive Design: Create a user-friendly interface that allows users to easily add, modify, and delete tasks and reminders.
* Task Visualization: Provide an organized view of tasks, allowing users to sort them by deadline or priority. Implement sorting functionality using the heapsort algorithm.

**ArrayList:**

* Implement an ArrayList to store the information of reminders and activities.

**Push Notification:**

* Notification Integration: Present popup messages or notifications in various formats based on the implemented code editor.

Step 4. Preliminary Designs (Discard Solutions Incompatible with the Model)

**Hash Table Design:**

* Hash Table Implementation: Based on a class or data structure representing the hash table. It should include methods for adding, deleting, and searching for tasks and reminders using the hash function.
* Collision Handling: Implementation of collision resolution based on the chosen strategy (chaining or open addressing).
* Hash Function Optimization: Ensuring that the hash function is efficient and evenly distributes keys (Validation).

**Heapsort Algorithm Design:**

* Heap Data Structure: Based on a heap data structure, allow efficient insertion of tasks and reminders as well as element extraction based on priority.
* Sorting: Implement the heapsort algorithm in a way that tasks can be sorted by deadline or priority according to user needs.

**Priority Queue Design:**

* Task Classification: Use a minimum priority queue (min heap) for prioritized tasks and a maximum priority queue (max heap) for non-prioritized tasks.
* Priority Updates: Allow users to change the priority of a task and adjust its position in the priority queue.

**User Interface (UI) Design:**

User Flow: Users will interact with a menu for various activities implemented in the system.Welcome to:

/—------**ALWAYS NOTE**—--------/

1. Add a new task.
2. Modify an existing task.
3. Delete a task.
4. View the list of tasks and reminders.
5. Filter tasks by priority.
6. Undo the last action.
7. Notification settings.
8. Exit the system.

**Push Notification Design:**

Notification Integration: Standardized computer-generated messages will be used by the application depending on the generated code. This will include configuring notifications and managing events that trigger notifications or the format in which they will be presented to the user. Implementation will not depend on a specific format or notice but rather on the code editor used.

**Undo System Design:**

Action Stack: A Last-In, First-Out (LIFO) stack will be designed to record user actions along with details of the affected tasks. Ensuring that the stack is memory-efficient.

**ArrayList:**

It does not relate practically to the exercise; therefore, this option is discarded.

Step 5. Testing and Solution Selection (Evaluate Precision, Efficiency, Complexity, Algorithmic Implementation Ease)

**Criterion A. Solution Precision:**

[2] Exact (an exact solution is preferred)

[1] Approximate

**Criterion B. Efficiency:**

[4] Constant

[3] Greater than constant

[2] Logarithmic

[1] Lineal

**Criterion C. Completeness:**

[3] All

[2] More than one if available, though not all

[1] Only one or none

**Criterion D. Ease of Algorithmic Implementation:**

[2] Compatible with basic arithmetic operations of a modern computer

[1] Not completely compatible with basic arithmetic operations of a modern computer

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Criterion A** | **Criterion B** | **Criterion C** | **Criterion D** | **TOTAL** |
| **Hash Table** | 2 | 3 | 3 | 2 | 10 |
| **Heapsort Algorthym** | 2 | 3 | 2 | 2 | 9 |
| **Priority Queue** | 2 | 4 | 2 | 2 | 10 |
| **UI** | 1 | 4 | 1 | 2 | 8 |
| **Push Notification** | 1 | 2 | 1 | 2 | 6 |
| **Undo** | 2 | 3 | 1 | 2 | 8 |
| **Arraylist** | 1 | 1 | 1 | 2 | 5 |

The choice of solutions is as follows:

Hash Table, Heapsort Algorthym, Priority Queue, UI, Push Notification, Undo.

Points to be developed after the entire research process:

**MaxHeapify:**

**Operations**:

**maxHeapify**(from): A method that adjusts the heap starting from a given node to maintain the maximum heap property. It ensures that the node at position from is greater than or equal to its children, thereby preserving the maximum heap property.

**getLeft**(from): A method that returns the position of the left child of the node at position from.

**getRight**(from): A method that returns the position of the right child of the node at position from.

**swap**(from, to): A method that swaps the elements at positions from and to in the heap.

**Attributes**:

**list**: A list (e.g., ArrayList) that stores the elements of the heap.

**heapSize**: The number of valid elements in the heap.

**HeapSort:**

**Operation**:

**heapSort**(): A method that sorts the elements stored in the heap in non-decreasing order (ascending order). It first builds a maximum heap from the elements, and then iteratively removes the maximum element from the heap, placing it in the appropriate position in the sorted array. The heap size is gradually reduced as elements are sorted.

Here's how you can include it in the TAD definition:

**Maximum Heap**

**Description**:

The "Maximum Heap" TAD, also known as "Max Heap," is a data structure that organizes its elements in such a way that the root element is always the maximum concerning the specified order. It is commonly used to implement priority queues and efficient sorting algorithms like heapsort.

**Operations**:

**maxHeapify**(from): A method that adjusts the heap starting from a given node to maintain the maximum heap property. It ensures that the node at position from is greater than or equal to its children, preserving the maximum heap property.

**getLeft**(from): A method that returns the position of the left child of the node at position from.

**getRight**(from): A method that returns the position of the right child of the node at position from.

**swap**(from, to): A method that swaps the elements at positions from and to in the heap.

**heapSort**(): A method that sorts the elements stored in the heap in non-decreasing order. It first builds a maximum heap from the elements, and then iteratively removes the maximum element from the heap, placing it in the appropriate position in the sorted array. The heap size is gradually reduced as elements are sorted.

**Attributes**:

**list**: A list (e.g., ArrayList) that stores the elements of the heap.

**heapSize**: The number of valid elements in the heap.

**Priority Queue**

A "Priority Queue" is a data structure that maintains a collection of elements with associated priorities. It allows elements to be inserted with an associated priority and provides operations to retrieve and remove elements based on their priority. Elements with higher priorities are dequeued before elements with lower priorities. A Priority Queue can be implemented as a max-heap, min-heap, or other data structures depending on the desired behavior.

**Operations**:

**insert**(element, priority): Inserts an element with the specified priority into the priority queue.

**peek**(): Returns the element with the highest priority (the element at the front of the queue) without removing it.

**dequeue**(): Removes and returns the element with the highest priority (the element at the front of the queue).

**isEmpty**(): Checks whether the priority queue is empty.

**size**(): Returns the number of elements currently in the priority queue.

**Attributes**:

**heap** (or data structure of choice): The data structure used to implement the priority queue. Depending on the implementation, it could be a max-heap or min-heap to support efficient peek and dequeue operations.

**size**: The number of elements currently in the priority queue.

**Stack**

A "Stack" is a linear data structure that follows the Last-In-First-Out (LIFO) principle, where the last element added to the stack is the first to be removed. Stacks are commonly used in various computer science applications, such as function call management, expression evaluation, and backtracking algorithms.

**Operations**:

**push**(element): Adds an element to the top of the stack.

**pop**(): Removes and returns the element from the top of the stack.

**peek**(): Returns the element from the top of the stack without removing it.

**isEmpty**(): Checks whether the stack is empty.

**size**(): Returns the number of elements currently in the stack.

**Attributes**:

**elements**: The collection of elements stored in the stack. It can be implemented as an array or a linked list, depending on the choice of data structure.

**Hash Table**A "Hash Table" is a data structure that stores key-value pairs and provides efficient insertion, retrieval, and deletion of values based on their associated keys. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Hash tables are widely used for fast data access.

**Operations**:

**put**(key, value): Inserts a key-value pair into the hash table. If the key already exists, the associated value is updated.

**get**(key): Retrieves the value associated with the given key. If the key is not found, an appropriate indication (e.g., null or an exception) is returned.

**remove**(key): Removes the key-value pair associated with the given key from the hash table.

containsKey(key): Checks if the hash table contains the specified key.

**size**(): Returns the number of key-value pairs in the hash table.

isEmpty(): Checks whether the hash table is empty.

Attributes:

buckets: An array (or linked list) of storage units (buckets or slots) used to store key-value pairs. The index of each bucket is determined by a hash function applied to the keys.

**size**: The number of key-value pairs currently stored in the hash table.

**Node Heap**

In a Heap data structure, a "Node" represents an element that is part of the heap. Each node typically has a key and an associated value (or payload). Heaps are often used for priority queues and to efficiently manage elements with respect to their priority (e.g., in a max-heap or min-heap).

**Attributes**:

**key**: The key associated with the node, which determines its position in the heap. In a max-heap, nodes with higher keys are placed higher in the tree, and in a min-heap, nodes with lower keys are placed higher.

**value**: The value or payload associated with the node. This is the data or information that the node holds.

**left**: A reference to the left child node in the binary heap.

**right**: A reference to the right child node in the binary heap.

**parent**: A reference to the parent node in the binary heap. The root node typically has a null parent.

**Methods**:

Nodes in a heap are typically used in the context of the heap data structure, and the methods for manipulating the heap (e.g., insert, delete, heapify) are applied to the entire heap rather than individual nodes. However, nodes play a crucial role in constructing and maintaining the heap structure.

**1. Heap Sort Algorithm:**

Temporal Complexity (Time Complexity):

Building the heap (buildHeap): The time complexity for this operation is O(n), where n is the number of elements in the input list.

Sorting the heap (heapSort): The time complexity for heap sort is O(n \* log(n)), which is an upper bound for sorting operations using a binary heap.

Spatial Complexity (Space Complexity):

The space complexity for heap sort is O(1) because it sorts the elements in-place, without using additional memory for data storage.

HeapSort(arr):

BuildHeap(arr) // Build a max-heap from the input array

n = arr.length

for i from n-1 to 1: // Traverse the heap from last to first element

Swap(arr[0], arr[i]) // Swap the root (max element) with the last element

MaxHeapify(arr, 0, i) // Restore the max-heap property

BuildHeap(arr):

n = arr.length

for i from n/2 down to 0: // Start from the middle of the array and go up

MaxHeapify(arr, i, n)

MaxHeapify(arr, i, n):

left = 2 \* i + 1

right = 2 \* i + 2

largest = i

if left < n and arr[left] > arr[largest]:

largest = left

if right < n and arr[right] > arr[largest]:

largest = right

if largest ≠ i:

Swap(arr[i], arr[largest])

MaxHeapify(arr, largest, n)

Swap(a, b):

temp = a

a = b

b = temp

**2. Hash Table Operations (Generic):**

Temporal Complexity (Time Complexity):

put(key, value): The time complexity for inserting a key-value pair into a hash table depends on the efficiency of the hash function and the collision resolution strategy. In the average case, it's O(1), but in the worst case (due to collisions), it can degrade to O(n), where n is the number of elements in the bucket.

get(key): Retrieving a value from a hash table also depends on the hash function and the collision resolution. In the average case, it's O(1), but in the worst case, it can be O(n).

remove(key): Removing a key-value pair from the hash table follows similar complexity as the get operation.

Spatial Complexity (Space Complexity):

The space complexity of a hash table mainly depends on the number of key-value pairs stored. In the average case, it's O(n), where n is the number of key-value pairs. Additional space may be required for the array or linked list buckets, which would contribute to space complexity as well.

**HashTable Operations:**

Put(key, value):

hash = HashFunction(key)

index = hash % array\_size

if collision occurs at index:

HandleCollision(index, key, value)

else:

Create a new key-value pair and insert it at index

Get(key):

hash = HashFunction(key)

index = hash % array\_size

if key found at index:

Return the associated value

else:

Return "Key not found"

Remove(key):

hash = HashFunction(key)

index = hash % array\_size

if key found at index:

Remove the key-value pair

else:

Return "Key not found"

HandleCollision(index, key, value):

Implement collision resolution strategy (e.g., open addressing, chaining)

Insert the key-value pair into the appropriate data structure