Integrating task
task management and reminder systems

DANIEL MEJIA | JEAN GALLEGO

Content

1. INTRODUCTION.

Efficient task and reminder management is essential for personal and professional productivity and success. To address this need, the development of a task and reminder management system has been proposed that will allow users to effectively add, organize and manage their daily to-dos.

This project aims to design and implement a comprehensive system that addresses all dimensions of task management, from the storage of information to the presentation of an intuitive user interface and advanced functionalities. The system will be based on efficient algorithms and data structures to ensure optimal performance.

1.1. Components and functionalities (SGTR)

1.1.1. Task storage and reminders

We will use a hash table to store tasks and reminders. Each entry will have a unique identifier as a key and the task/reminder information as a value. This information includes the title, description, due date, and priority.

1.1.2. User Interface

We will design a user interface that allows users to add, modify, and delete tasks and reminders. Users will be able to see a list of all tasks and reminders, sorted by due date or priority.

1.1.3. Priority Management

There will be two categories of tasks: "priority" and "non-priority".

- For priority tasks, we will use a priority queue to organize them according to their importance. When a user adds a new priority task, it will be added to the queue according to its importance, which will ensure that important tasks are handled first.
- Non-priority tasks are handled on a first-in-first-out (**FIFO**) basis.

1.1.4. Undo function

We will implement a method to undo the actions performed by a user in the system. We will use a stack (**LIFO**) to keep track of the actions performed. The general process for the "Undo" function will be:

1. Create an action stack to keep track of the user's actions.
2. Record each action performed by the user in the stack, including details of the action and the task affected.
3. Implement a last action undo method that unstacks the last action and undoes the corresponding action based on the information stored in the stack.
4. Provide the user interface with an "Undo" option to undo the last action performed.

With these features, our task and reminder management system will be efficient and allow users to organize and manage their tasks effectively.

2. REQUIREMENTS ANALYSIS.

1.2.Case study:

Client	**Marlon Gomez**
User	The system's end-users will consist of individuals seeking to efficiently manage their daily tasks and reminders.
Functional Requirements	**R1. Add Tasks**
	R2. Modify Tasks
	R3. Remove Tasks
	R4. Task Classification
	R5. Sort Tasks
	R6. Undo Actions
Non-Functional Requirements	Backup and Recovery
	Browser Compatibility
	scalability
	Data Security

2.1.1. Functional Requirement N°1

<table>
<tr><th colspan="4">R1</th></tr>
<tr><td>Name</td><td colspan="3">Add Task</td></tr>
<tr><td>overview</td><td colspan="3">Allow users to add new tasks by specifying title, description, due date, and priority.</td></tr>
<tr><td rowspan="2">Inputs</td><td>Input Name</td><td>Data Type</td><td>VVC</td></tr>
<tr><td>taskDetails</td><td>String (Title and Description), Date (Due Date), Integer (Priority)</td><td>Title and Description should not be empty, Due Date should be in the future, Priority should be within a defined range.</td></tr>
<tr><td colspan="4">Result or Postcondition</td></tr>
<tr><td rowspan="2">Outputs</td><td>Output Name</td><td>Data Type</td><td>Format</td></tr>
<tr><td>taskAdded</td><td>Boolean</td><td>True if the task is added successfully, False if there is an error.</td></tr>
</table>

2.1.2. Functional Requirement N°2

R2			
Name	**Modify Tasks**		
overview	Allow users to edit existing tasks to update information such as title, description, or due date.		
Inputs	***Input Name***	***Data Type***	***VVC***
	updateTaskDetails	String (Title and Description), Date (Due Date)	Title and Description should not be empty, Due Date should be in the future, Priority should be within a defined range.
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	taskAdded	Boolean	True if the task is modified successfully, False if there is an error.

2.1.3. Functional Requirement N°3

R3			
Name	**Delete Tasks**		
overview	Allow users to delete tasks that are no longer relevant.		
Inputs	***Input Name***	***Data Type***	***VVC***
	taskID	Integer	Task ID should correspond to an existing task.
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	taskDeleted	Boolean	True if the task is deleted successfully, False if there is an error.

2.1.4. Functional Requirement N°4

R4			
Name	**Categorize Tasks**		
overview	Allow users to categorize tasks as "Priority" or "Non-priority" when creating them.		
Inputs	***Input Name***	***Data Type***	***VVC***
	taskCategory	String	Should be either "Priority" or "Non-priority"
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	taskCategorized	Boolean	True if the task is deleted successfully, False if there is an error.

1.1.1. Functional Requirement N°5

R5			
Name	**Sort Tasks**		
overview	Allow users to view a list of all their tasks, sorted by due date or priority.		
Inputs	***Input Name***	***Data Type***	***VVC***
	sortingOption	String	Should be either "Due Date" or "Priority"
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	taskList	List Of Tasks	Display the tasks in the selected sorting order.

2.1.5. Functional Requirement N°6

R6			
Name	**Undo Actions**		
overview	Implement a feature that allows users to undo the last action performed in the system.		
Inputs	***Input Name***	***Data Type***	***VVC***
	undoRequest	*N/a*	*N/a*
Result or Postcondition			
Outputs	***Output Name***	***Data Type***	***Format***
	actionUndone	*Boolean*	True if the last action is successfully undone, False if there are no actions to undo or if there is an error.

3. ANALYSIS.

In computer science, algorithm analysis is an essential process for understanding the performance and efficiency of algorithms used to solve problems. Two critical aspects of this analysis are time complexity and space complexity. Time complexity refers to the time an algorithm takes to complete its task as a function of input size, while space complexity refers to the amount of memory or storage space an algorithm uses as a function of input size.

In this section, we will perform a detailed analysis of the time and space complexity of at least two algorithms implemented in this project. This analysis will allow us to evaluate and compare the performance of these algorithms in terms of efficiency and resources used. By better understanding the complexity of our algorithms, we will be able to make informed decisions about which is the best choice for a given application.

1.3. Implemented algorithm time complexity analysis.

During this analysis, we will break down each line of code within the put and get methods and evaluate the individual time complexity of each method. To fully understand the performance of these methods, we will identify the key operations that influence execution time and consider both best and worst cases.

3.1.1. Hash tables - Put Method

```
public void put(K key, V value){

    int index = hash(key); // O(1) - hash index calculation.
    table[index] = new HTNode<>(key, value); // O(1) - Assignment to the hash table
    size++; // O(1) - Increase in size.

}
```

3.1.1.1. Key operations.

1. **Calculate the hash index:** The first key operation is to compute the hash index of the key. This is done using a hash function, which is generally of constant time, resulting in **O(1).**
2. **Assignment to the hash table:** The next operation is to map the value in the hash table to the index calculated in the previous step. This is a constant time operation. **O(1).**
3. **Size increment:** After inserting the element into the hash table, the size of the data structure (size) is incremented. This operation is also time constant. **O(1).**

3.1.1.2. Best case:

In the best case, when there are no collisions (each key maps to a unique index), all these operations are constant time (O(1)).

3.1.1.3. Worst case:

In the worst case, when there are collisions and multiple keys must map to the same hash index, the execution time may increase due to the index lookup and the possibility of reassignment. In the worst case, the complexity could be O(n), where n is the number of colliding elements in each index.

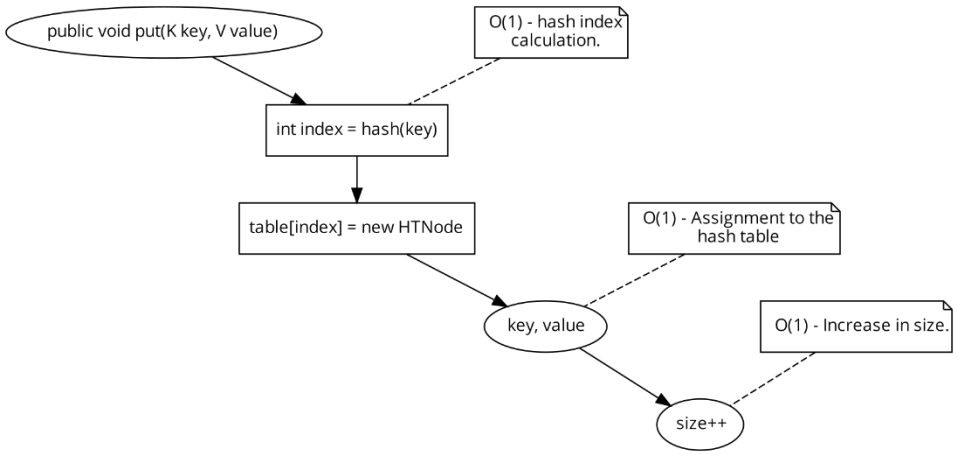

Figure 1 flowchart

3.1.2. Hash tables – Get Method.

```
public V get(K key) {

    V value = null; // O(1) - Initialization of value.
    int i = hash(key); // O(1) - Hash index calculation.
    if(table[i] != null && table[i].getKey().equals(key)){// O(1) - Checking and accessing the value.

        value = table[i].getValue(); // O(1) - Access to the value.
    }

    return value;
}
```

3.1.2.1. Key operations.

1. **Calculate the hash index:** The first key operation is to compute the hash index of the key. This is done using a hash function, which is generally of constant time, resulting in **O(1).**

2. **Check and access the value:** The next step is to check if the computed index contains an element and if the key in this element matches the given key. If the match is successful, the associated value is accessed. This check and access is time constant, **O(1).**

3.1.2.2. Efficiency

The efficiency of the get method depends on the distribution of keys and the implementation of the hash function used. In general, the get method tends to be efficient for most searches but may be affected by collisions in exceptional situations.

1.4.Implemented algorithm space complexity analysis.

In this analysis, we will examine in detail the amount of memory used by the methods . We will evaluate memory usage in terms of constant memory (fixed memory) and additional memory (memory that varies according to inputs and operations). To do this, we will look at the data structures used and how they are handled in each method.

3.1.3. Task Manager – Add Task Method

```
public String addTask(int key, int id, String title, String desc, String limitDate, int prio){
    String msg = ""; // Constant space O(1)

    if(taskTable.getSize() != 10){ // Constant space O(1) (assuming getSize is O(1))
        Priority priority = null; // Constant space O(1)

        if(prio == 1){ // Constant space O(1)
            priority = Priority.PRIORITY; // Constant space O(1)
        }
        else if(prio == 2){ // Constant O(1) space.
            priority = Priority.NONPRIORITY; // Constant space O(1)
        }

        Task newTask = new Task(id, title, limitDate, desc, priority); // Additional space to create a Task object.

        taskTable.put(key, newTask); // Additional space to add an element to the hash table

        if(priority == Priority.PRIORITY){ // Constant space O(1)
            priorityQueue.enqueue(newTask); // Additional space to add an item to the priority queue
        }
        else if(priority == Priority.NONPRIORITY){ // Constant O(1) space.
            nonPrioQueue.enqueue(newTask); // Additional space to add an item to the non-priority queue
        }

        msg = "Task added."; // Constant O(1) space.
        actions.push(new ActionRecord(key,newTask, Action.ADD)); // Additional space to add an item to the stack
    }
    else{
        msg = "Table full!"; // Constant space O(1)
    }

    return msg; // Constant O(1) space.
}
```

3.1.3.1. Analysis Memory usage.

Constant space:

local variables such as msg, priority, newTask, newDesc, newDate, and other local variables.

Additional space:

- taskTable (HashTable): The amount of space required depends on the size of the hash table. In this case, a hash table is created with an initial capacity of 10, so the additional space depends on this capacity.
- PriorityQueue: The space required depends on the number of items in the priority queue at the time.
- nonPrioQueue: The amount of space required depends on the number of items in the non-priority queue at the time.
- actions (stack): The space required depends on the number of action records stored in the stack.

3.1.4. Task Manager – Undo Method

```
public String undo() {
    String msg = "";  // Constant space O(1)
    ActionRecord action = actions.pop();  // Constant space O(1) for pop


    if(action != null) {  // Constant space O(1) for checking if action is not null


        if (action.getType() == Action.ADD) {  // Constant space O(1)
            Task addedTask = action.getTask();  // Constant space O(1)
            int addedTKey = action.getTKey();  // Constant space O(1)
            taskTable.remove(addedTKey);  // Additional space to remove an element
            msg = "Add action undone.";  // Constant space O(1)
        } else if (action.getType() == Action.MODIFY) {  // Constant space O(1)
            int modifiedTKey = action.getModifiedTKey();  // Constant space O(1)
            String modifiedDesc = action.getModifiedDesc();  // Constant space O(1)
            String modifiedDate = action.getModifiedDate();  // Constant space O(1)
            Task modifiedTask = taskTable.get(modifiedTKey);  // Constant space O(1)
            modifiedTask.setDesc(modifiedDesc);  // Constant space O(1)
            modifiedTask.setLimitDate(modifiedDate);  // Constant space O(1)
            msg = "Modify action undone.";  // Constant space O(1)
        } else if (action.getType() == Action.REMOVE) {  // Constant space O(1)
            Task removedTask = action.getTask();  // Constant space O(1)
            int removedTKey = action.getTKey();  // Constant space O(1)
            taskTable.put(removedTKey, removedTask);  // Additional space to put an element
            msg = "Remove action undone.";  // Constant space O(1)
        }
    } else {
        msg = "No actions recorded.";  // Constant space O(1)
    }
    return msg;  // Constant space O(1)
}
```

3.1.4.1. Analysis Memory usage.

Constant space:

Local variables such as msg, action, addedTask, addedTKey, modifiedTKey, modifiedDesc, modifiedDate, removedTask, removedTKey, and other local variables.

Additional space:

Depending on the action being undone, hash table elements and queues may be added or removed. The additional space depends on the specific action being undone.

---

s