SMART COMMAND MANAGER

PROJECT REPORT

Contents

[1.Problem statement overview 3](#_Toc202275084)

[2.thought process and approach 4](#_Toc202275085)

[3.data structures used 5](#_Toc202275086)

[4.time and space complexity analysis 6](#_Toc202275087)

[5.how to run application 7](#_Toc202275088)

[6.sample input output commands 8](#_Toc202275089)

[7.flow chart 9](#_Toc202275090)

# 1.Problem statement overview

In many real-life systems — like task managers, operating systems, or collaborative apps — there’s a constant need to manage, execute, and track tasks in the right order. Some tasks are more important than others, and sometimes you need to reverse an action or redo something you just undid. That’s where the **Smart Command Manager** comes in.

This project focuses on building a simple but intelligent system that lets users add commands, give them priority if needed, execute them one by one, and even undo or redo those actions. The goal is to provide full control over the command flow in a clean and efficient way.

To make the system fast and reliable, the project uses well-known data structures:

* **Queue** for normal task ordering
* **Priority Queue** for urgent tasks
* **Stack** to manage undo/redo history

The Smart Command Manager doesn’t just execute tasks — it also displays pending and completed ones clearly, helping users stay organized and error-free. It’s a powerful example of how data structures can be used together to solve real-world workflow problems in a smooth and structured way.

# 2.thought process and approach

In modern software systems, tasks often need to be managed dynamically — with the ability to prioritize, undo, or redo actions as needed. Our goal was to create a **Smart Command Manager** that handles these operations efficiently, while staying simple and reliable.

We focused on:

* **Efficiency**: Using the right data structures (queues, heaps, and stacks) to make command operations fast and smooth.
* **Simplicity**: Designing a clear command-line interface so users can interact with the system without confusion.
* **Control**: Supporting undo and redo features through a well-managed command history, giving users flexibility over past actions.
* **Modularity**: Structuring the code into clean, logical parts for easy maintenance and future updates.

Python was selected for its readability and built-in support for data structures.

# 3.data structures used

**priority queue(heap)**

We use a **min-heap** to always pick the command with the **highest priority (smallest number)** first. It's perfect for auto-sorting tasks that must be executed in priority order.

**Stack (undo/redo)**

We store executed commands in a **stack** so we can easily undo the last action — just like going one step back in a browser. Redo uses another stack to bring it back.

**Dictionary lookup**

We use a **dictionary** to store command IDs and their text for **quick access** — it’s way faster than searching a list every time.

**Enumerate and sorting**

helps us show serial numbers (like 1, 2, 3...) alongside commands. Sorting keeps them ordered by priority, so the table looks clean and logical.

**ANSI color coding**

We add **colors and bold text** using special terminal codes — making output easier to read, like red for some danger zone e.g deletion and green for executed.

**Modular design with separate files**

Each feature (execution, undo, view, etc.) is in its **own Python file**, making the code clean, organized, and easier to maintain or upgrade.

**Input validation and error handling**

We check every user input — like only allowing numbers or handling out-of-range options — so the program doesn’t crash and feels smooth to use.

# 4.time and space complexity analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Dictionary (dict)** | **Queue - list based** | **Queue - deque based** |
| **Access by Key (e.g., dict[k])** | 🟢 **O(1)** average | 🔴 **O(n)** (search loop) | 🔴 **O(n)** (search loop) |
| **Insert (Add)** | 🟢 O(1) | 🟢 O(1) (append) | 🟢 O(1) (append) |
| **Delete by Key** | 🟢 O(1) | 🔴 O(n) (search + remove) | 🔴 O(n) (search + remove) |
| **Pop First Item** | ❌ N/A (unordered) | 🔴 O(n) (slow in list) | 🟢 O(1) (fast in deque) |
| **Search by Value** | 🔴 O(n) | 🔴 O(n) | 🔴 O(n) |
| **Space Complexity** | 🟢 O(n) | 🟢 O(n) | 🟢 O(n) |

**O(1) – Constant Time 🟢**

The operation takes the same amount of time no matter how much data there is.

**O(n) – Linear Time 🔴**

The time grows directly with the number of elements.

# 5.how to run application.

### **Requirements:**

* Python 3.x installed on your system.
* Required libraries:
  + ***pyfiglet***(for stylized title)
  + ***heapq*** (built-in, no install needed)
  + ***datetime*** (built-in, no installation needed)
  + ***tabulation(****for tables printing)*
  + ***emoji****(for printing various emojis on terminal)*

You can install required libraries by running:

*pip install pyfiglet*

*pip install tabulation*

*pip install emoji*

### **Steps to Run (Command-Line Version):**

1. Open **Command Prompt** (CMD).
2. Navigate to the project folder directory in your computer :

For example: *cd D: \DSA\projects\project2*

1. Run the Python script:

*Python main.py*

1. The program will launch in terminal.
2. Follow on-screen instructions to use the inventory system.

# 6.sample input output commands

# 7.flow chart

**START PROGRAM**

**DISPLAY TITLE**

**SMART COMMAND MANAGER**

**DISPLAY MENU**

**1–ADD COMMAND 2–EXECUTE**

**3–DISPLAY TABLE 4–DELETE**

**5-PENDING COMMANDS**

**6–EXECUTED COMMANDS**

**7-STATUS OF COMMANDS**

**8- EXIT PROGRAM**

**8–EXIT**

**ADD: ADD COMMAND TO TABLE**

**DELETE: DELETE COMMAND**

**UNDO AND REDO FUNCTIONS**

**STATUS: COMMANDS STATUS**

**CLEAR: EMPTY ARRAY**

**DISPLAY: DISPLAY TABLE**

**LOOP OR EXIT(8)**

**PERFORM ACTION BASED ON**

**INPUT (USING OOP CLASS)**