DATABASE MANAGEMENT SYSTEMS LAB

REPORT ASSIGNMENT-1

NAME: SOHAM LAHIRI

CLASS: BCSE UG-III 6TH SEMESTER

ROLL NO: 002210501107

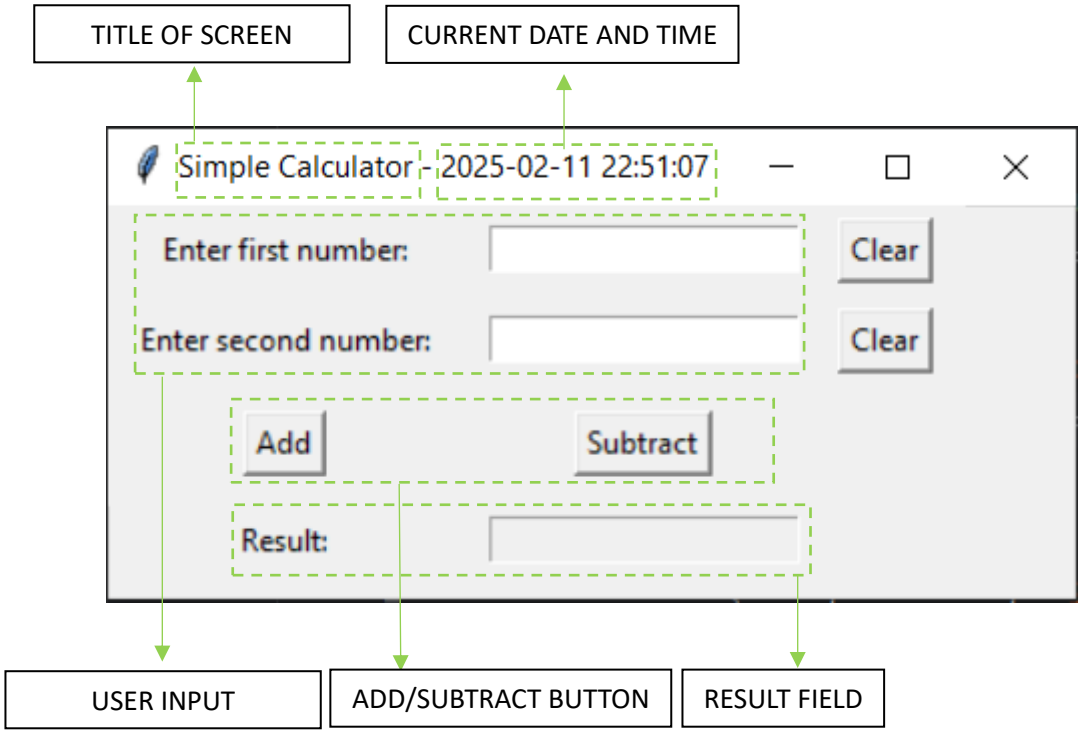
GROUP: A3

SUBMISSION DATE: 11/02/2025

Problem Statement:

Develop an application as follows.

User will enter two numbers. Provide button to add or to find the difference. Depending on the option result will be shown in result box. Add a suitable title to the screen. Whenever the form is loaded, also display current date in the title bar.



Simple Calculator using Tkinter in Python

The Simple Calculator is a graphical user interface (GUI) application built using Python's Tkinter library. It provides basic arithmetic operations—addition and subtraction—allowing users to input two numbers and compute the results interactively. The application also features an updating window title displaying the current date and time.

Features:

1. **Real-Time Clock:**
    - The window title dynamically updates every second to display the current date and time.
2. **User Input Fields:**
    - Two entry fields allow users to input numbers.
    - "Clear" buttons are provided for each entry field to easily reset input values.

3. **Mathematical Operations:**
    - Users can perform addition and subtraction by clicking the respective buttons.
    - The result is displayed in a read-only text field.
4. **Error Handling:**
    - If the user inputs non-numeric values, an error message is displayed in red.
    - The error message clears when a valid input is entered.

Components:

1. **Label:**
    - Labels for "Number 1", "Number 2", "Result", "Error Label", and "Title Label".
2. **Button:**
    - "Clear" buttons for resetting inputs.
    - "Add" and "Subtract" buttons for performing operations.
3. **TextField:**
    - Input fields for "Number 1" and "Number 2".
    - Read-only text field for displaying the "Result".

Implementation Details:

1. GUI Initialization

Code:

```
import tkinter as tk
from datetime import datetime

def update_title():
    current_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    root.title(f"Simple Calculator - {current_time}")
    root.after(1000, update_title)
```

- The application is initialized using tk.Tk() to create the main window (root).
- The update_title function updates the window title every second using root.after(1000, update_title), ensuring real-time clock updates.

2. User Input Handling

Code:

```
entry1 = tk.Entry(root)
entry2 = tk.Entry(root)
```

- Two tk.Entry widgets collect numerical inputs from the user.

3. Arithmetic Operations (Addition & Subtraction)

Code:

```
def calculate(operation):
    try:
        num1 = float(entry1.get())
        num2 = float(entry2.get())
        error_label.config(text="")

        if operation == "add":
            result.set(num1 + num2)
        elif operation == "subtract":
            result.set(num1 - num2)
    except ValueError:
        error_label.config(text="Error: Invalid input. Please enter numeric values.",
fg="red")
        result.set("")
```

- Retrieves input values and converts them to float.
- Performs the chosen arithmetic operation.
- Displays the result or an error message in case of invalid input.

4. Clear Input Fields

Code:

```
def clear_entry(entry):
    entry.delete(0, tk.END)
```

- Clears the specified entry field when the "Clear" button is clicked.

5. Error Handling

- **Invalid Input Handling:**
    - If non-numeric values are entered, an error message appears.
    - Example: error_label.config(text="Error: Invalid input. Please enter numeric values.", fg="red")

6. UI Layout

Code:

```
label1 = tk.Label(root, text="Enter first number:")
label1.grid(row=0, column=0, padx=10, pady=5)

entry1 = tk.Entry(root)
entry1.grid(row=0, column=1, padx=10, pady=5)
```

```
clear_button1 = tk.Button(root, text="Clear", command=lambda: clear_entry(entry1))
clear_button1.grid(row=0, column=2, padx=5, pady=5)
```

- Labels, buttons, and entry fields are arranged using grid().
- Components include "Add", "Subtract", "Clear" buttons, input fields, and result display.

7. Event Loop Execution

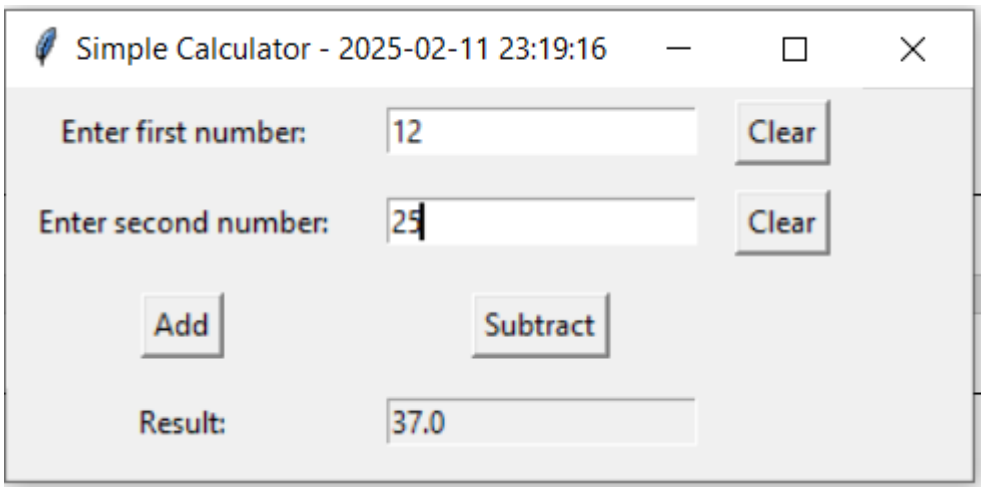
Code:

```
root.mainloop()
```

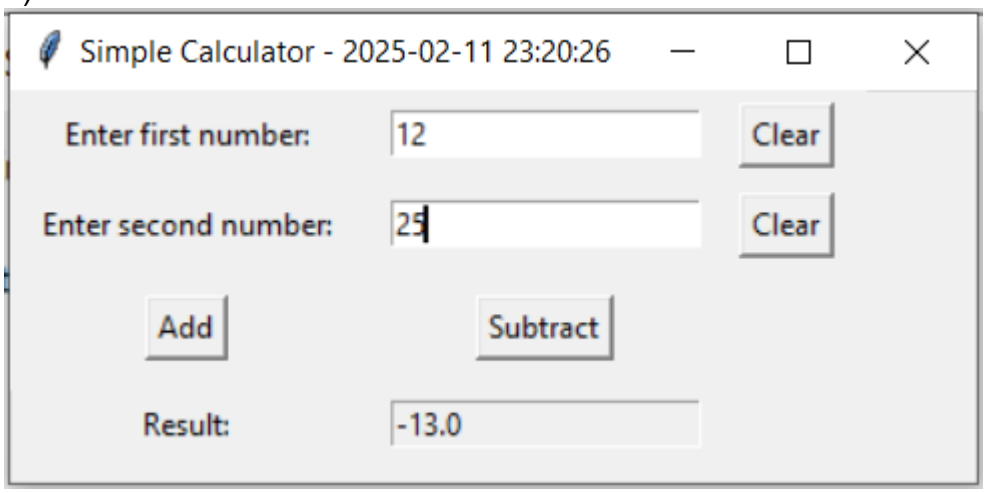
- The application runs continuously, allowing user interactions.

OUTPUT:

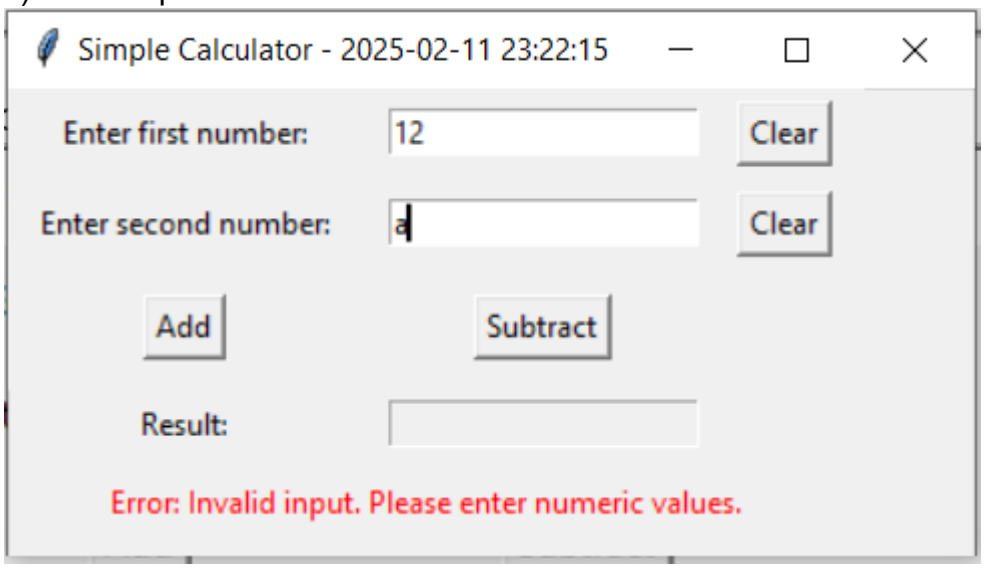
a) Addition



b)Subtraction



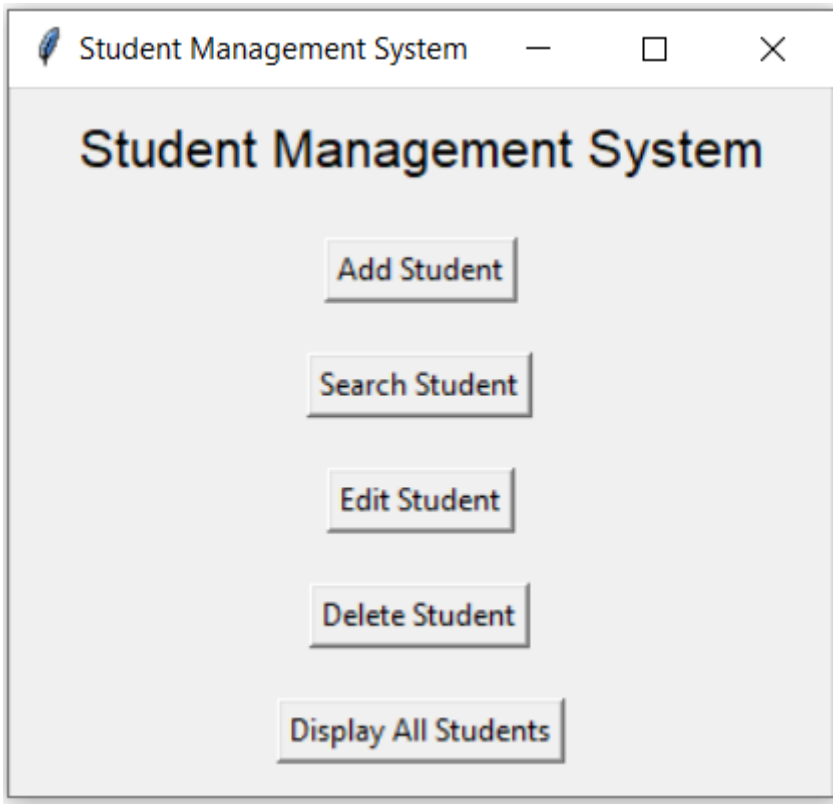
c)Invalid Input Error



Problem Statement:

Consider list of departments (dept code and name), list of students (roll, dept code, name, address and phone) preloaded in array. Now develop an application for the following.
User may add/search/edit/delete/display all student record. While adding, ensure roll must be unique, a list of dept name to be shown from which user selects one and corresponding dept code to be stored. On collecting the data user may choose CANCEL/SAVE button to decide course of action. For searching user provides roll. If it exists details are shown else suitable message to be displayed. To delete user provides roll. If it does not exist then suitable message is to be displayed. To edit also user provides roll. If it exists user may be allowed to edit any field except roll. User may select CANCEL/SAVE to decide course of action. To display all records, at a time five records are to be shown. IT will also have PREV/NEXT button to display previous set and next set respectively. When first set is displayed PREV button must be disabled and at last set NEXT button must be disabled.

INTERFACE:



The **Student Management System** is a Python-based GUI application built using **Tkinter**. It allows users to **search, edit, delete, display, and filter student records**. The system also features **pagination**, enabling easy navigation of student records when dealing with large datasets.

GUI Components and Their Purpose:

1. Tk() - This initializes the main application window where all other UI elements are placed. It serves as the root container for the graphical user interface.
2. Label() - Used to display static text such as titles, field names, and messages. For example, labels are used to indicate input fields like "Name," "Roll Number," and "Department."
3. Entry() - This allows users to input data. It is used for fields like student name, address, phone number, and roll number. The values from these fields are later retrieved for processing student records.
4. Button() - Buttons are used to trigger actions. For example, an "Edit" button saves changes made to a student's details, while a "Delete" button removes a student record from the system. Navigation buttons such as "Next" and "Previous" allow the user to move between different pages of student records.
5. OptionMenu() - This dropdown menu is used to filter students by department. It allows users to select a specific department, which then updates the displayed student list accordingly.
6. Frame() - Frames are used to organize the UI into different sections. For instance, a frame is used to contain the list of students, making it easier to update the display dynamically without affecting other UI elements.
7. messagebox.showinfo() - This function displays popup messages to inform the user about the success or failure of an operation. For example, after successfully deleting a student, a message box appears confirming the deletion.

Functionalities:

1. Editing Student Records

Code:

```
def save_student(self, roll_var, name_var, address_var, phone_var, dept_var):
    roll = roll_var.get()
    name = name_var.get()
    address = address_var.get()
    phone = phone_var.get()
    dept_name = dept_var.get()

    if not roll or not name or not address or not phone or not dept_name:
        messagebox.showerror("Error", "All fields are required!")
        return

    if any(student["roll"] == roll for student in self.students):
        messagebox.showerror("Error", "Roll number must be unique!")
        return

    dept_code = next((dept["code"] for dept in self.departments if dept["name"] ==
dept_name), None)
```

```
    if not dept_code:
        messagebox.showerror("Error", "Invalid department selected!")
        return

    self.students.append({
        "roll": roll,
        "name": name,
        "address": address,
        "phone": phone,
        "dept_code": dept_code,
    })
    messagebox.showinfo("Success", "Student added successfully!")
    self.create_main_menu()
```

Purpose:
This function updates an existing student record by modifying the student's:

- **Name**
- **Address**
- **Phone number**
- **Department**

Process:

1. Retrieves the updated values from Tkinter **entry fields** (name_var, address_var, phone_var, dept_var).
2. Searches for the **department code** based on the department name entered by the user.
3. If a valid department is found, it updates the student's dept_code.
4. If an invalid department is entered, it displays an **error message**.
5. If the update is successful, a **success message** is shown.
6. Finally, it **redirects the user to the main menu**.

Key Validation:
1. Ensures the selected **department exists** before updating.
2. Prevents **invalid department selections** by showing an error message.

2. Deleting Student Records

Code:

```
def delete_student_window(self):
    self._clear_window()
    tk.Label(self.root, text="Delete Student", font=("Arial", 16)).pack(pady=10)

    roll_var = tk.StringVar()

    tk.Label(self.root, text="Enter Roll:").pack()
```

```
    tk.Entry(self.root, textvariable=roll_var).pack()

    tk.Button(self.root, text="Delete", command=lambda:
self.delete_student(roll_var)).pack(pady=5)
    tk.Button(self.root, text="Cancel",
command=self.create_main_menu).pack(pady=5)
```

Purpose:
This function creates a **delete window**, where the user can enter a student's **roll number** to delete their record.

Process:

1. Clears the current window using _clear_window().
2. Displays a label "Delete Student".
3. Provides an **entry field** for the user to input the **roll number**.
4. Includes two buttons:
    - **"Delete"** (calls delete_student())
    - **"Cancel"** (returns to the main menu)

Code:

```
def delete_student(self, roll_var):
    roll = roll_var.get()
    student = next((student for student in self.students if student["roll"] == roll), None)

    if not student:
        messagebox.showinfo("Info", "Student not found!")
        return

    self.students.remove(student)
    messagebox.showinfo("Success", "Student deleted!")
    self.create_main_menu()
```

Purpose:
Deletes a student record based on the roll number entered.

Process:

1. Retrieves the roll number from the input field (roll_var).
2. Searches for the student in the self.students list.
3. If the student exists, they are removed from the list.
4. Displays a **confirmation message** if deletion is successful.
5. If the student is not found, an **info message** is shown.
6. Redirects to the **main menu** after completion.

Key Validation:
1. Prevents accidental deletion by confirming the student exists.
2. Updates the record list dynamically

3. Displaying Student Records

Code:

```
def display_students_window(self):
    """Display all student records with pagination and filtering by department."""
    self._clear_window()
    tk.Label(self.root, text="Student Records", font=("Arial", 16)).pack(pady=10)

    # Department filter
    dept_var = tk.StringVar()
    dept_var.set("All Departments")  # Default value
    department_names = ["All Departments"] + [dept["name"] for dept in
self.departments]

    tk.Label(self.root, text="Filter by Department:").pack()
    dept_menu = tk.OptionMenu(self.root, dept_var, *department_names,
                              command=lambda d: self.update_display_by_department(d))
    dept_menu.pack(pady=5)

    # Frame to hold the student list
    self.student_list_frame = tk.Frame(self.root)
    self.student_list_frame.pack()

    # Navigation buttons for pagination
    nav_frame = tk.Frame(self.root)
    nav_frame.pack(pady=10)
    self.prev_button = tk.Button(nav_frame, text="Previous",
command=self.prev_page)
    self.next_button = tk.Button(nav_frame, text="Next", command=self.next_page)

    self.prev_button.grid(row=0, column=0, padx=5)
    self.next_button.grid(row=0, column=1, padx=5)

    # Back button
    tk.Button(self.root, text="Back to Menu",
command=self.create_main_menu).pack(pady=10)

    # Initial display
    self.update_display_by_department("All Departments")
```

Purpose:

Displays all student records with **pagination** and **department-wise filtering**.

Process:

1. **Clears the window** (_clear_window()).
2. Displays a **title label**: "Student Records".

3. Provides a **dropdown menu** (OptionMenu) to filter students by **department**.
4. Creates a **frame** (self.student_list_frame) to display the student list.
5. Adds **navigation buttons** ("Previous" and "Next") for **pagination**.
6. Adds a **Back to Menu** button to return to the main menu.
7. Calls update_display_by_department("All Departments") to load students.

Key Validation:
1. Allows **filtering by department**.
2. Supports **pagination** for large datasets.

Code:
```
def update_display_by_department(self, selected_dept):
    """Update the displayed student records based on department filter."""
    for widget in self.student_list_frame.winfo_children():
        widget.destroy()

    # Filter students by selected department
    if selected_dept == "All Departments":
        filtered_students = self.students
    else:
        filtered_students = [s for s in self.students if next(
            dept["name"] for dept in self.departments if dept["code"] == s["dept_code"])
== selected_dept]

    # Store filtered students and reset pagination
    self.filtered_students = filtered_students
    self.current_page = 0

    self.display_students_page()
```

Purpose:
Filters student records based on the selected **department**.

Process:
1. Clears the existing student display (self.student_list_frame).
2. Filters the student list based on the selected department.
    - If "All Departments" is selected, it displays all students.
    - Otherwise, it retrieves students whose dept_code matches the selected department.
3. Stores the **filtered students** list and resets the page to 0.
4. Calls display_students_page() to show the first page of results.

Key Validation:
1. Ensures only **students from the selected department** are displayed.
2. Automatically updates the **pagination system**.

Code:

```
def display_students_page(self):
    """Display students based on the current page."""
    for widget in self.student_list_frame.winfo_children():
        widget.destroy()

    start_idx = self.current_page * self.records_per_page
    end_idx = start_idx + self.records_per_page
    students_to_display = self.filtered_students[start_idx:end_idx]

    # Table headers
    headers = ["Roll", "Name", "Address", "Phone", "Department"]
    for col, header in enumerate(headers):
        tk.Label(self.student_list_frame, text=header, font=("Arial", 12, "bold"),
borderwidth=1, relief="solid",
                 width=15).grid(row=0, column=col)

    # Display student records
    for row, student in enumerate(students_to_display, start=1):
        dept_name = next(dept["name"] for dept in self.departments if dept["code"] ==
student["dept_code"])
        values = [student["roll"], student["name"], student["address"], student["phone"],
dept_name]

        for col, value in enumerate(values):
            tk.Label(self.student_list_frame, text=value, borderwidth=1, relief="solid",
width=15).grid(row=row,
                                                                    column=col)

    # Update navigation buttons
    self.prev_button.config(state=tk.NORMAL if self.current_page > 0 else
tk.DISABLED)
    self.next_button.config(state=tk.NORMAL if end_idx < len(self.filtered_students)
else tk.DISABLED)
```

Purpose:

Displays student records **page by page** using pagination.

Process:

1. Clears the student list display (self.student_list_frame).
2. Calculates the **start and end indices** based on the current page.
3. Displays **table headers**:
    - Roll Number
    - Name
    - Address
    - Phone

    - Department
4. Loops through the students for the current page and displays them in a **table format**.
5. Updates the **"Previous" and "Next"** buttons based on page availability.

Key Validation:
1. Prevents excessive scrolling by limiting students per page.
2. Maintains a **structured table format** for easy reading.

4. Pagination System

Code:
```
def next_page(self):
        """Move to the next page of student records."""
    if (self.current_page + 1) * self.records_per_page < len(self.filtered_students):
        self.current_page += 1
        self.display_students_page()
```

- Moves to the **next page** if more records exist.
- Updates self.current_page and refreshes the student list.

Code:
```
def prev_page(self):
    """Move to the previous page of student records."""
    if self.current_page > 0:
        self.current_page -= 1
        self.display_students_page()
```

- Moves to the **previous page** if applicable.
- Updates self.current_page and refreshes the student list.

Ensures smooth navigation between pages.
Prevents navigation beyond the available pages.

5 Clearing the Window

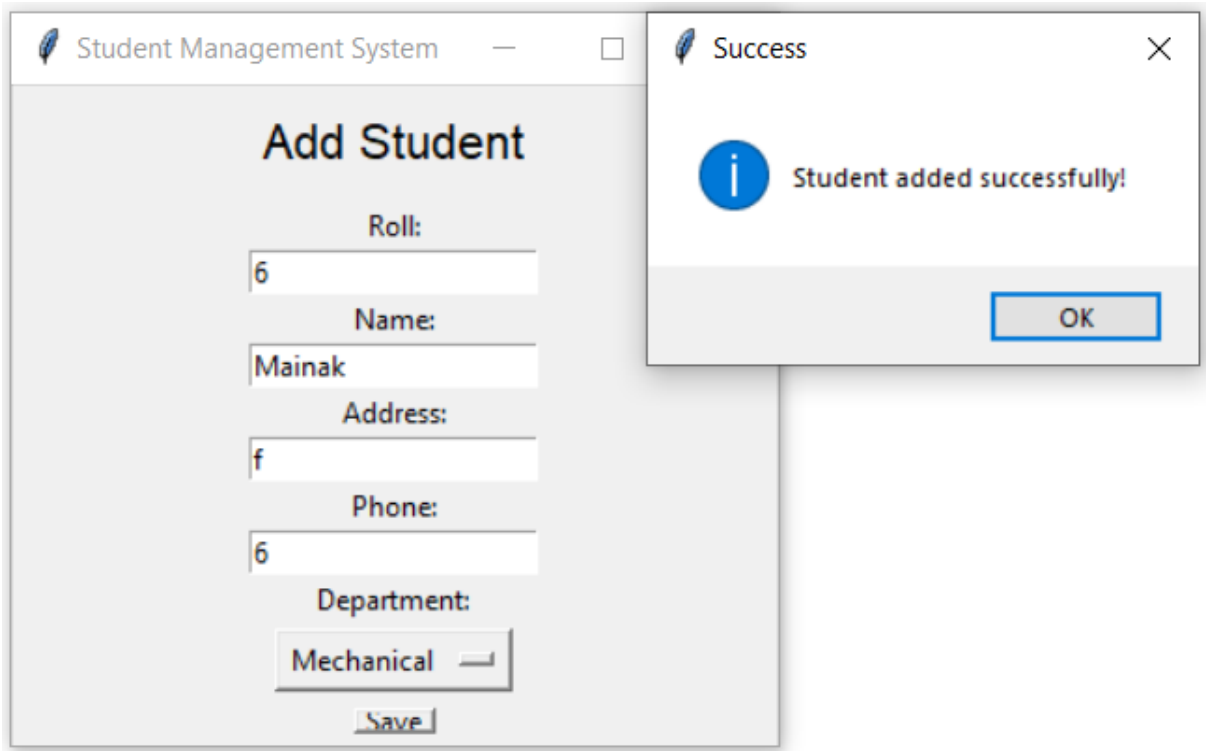
Code:
```
def _clear_window(self):
    for widget in self.root.winfo_children():
        widget.destroy()
```

- **Purpose:**
    - Removes all existing widgets before rendering a new UI.

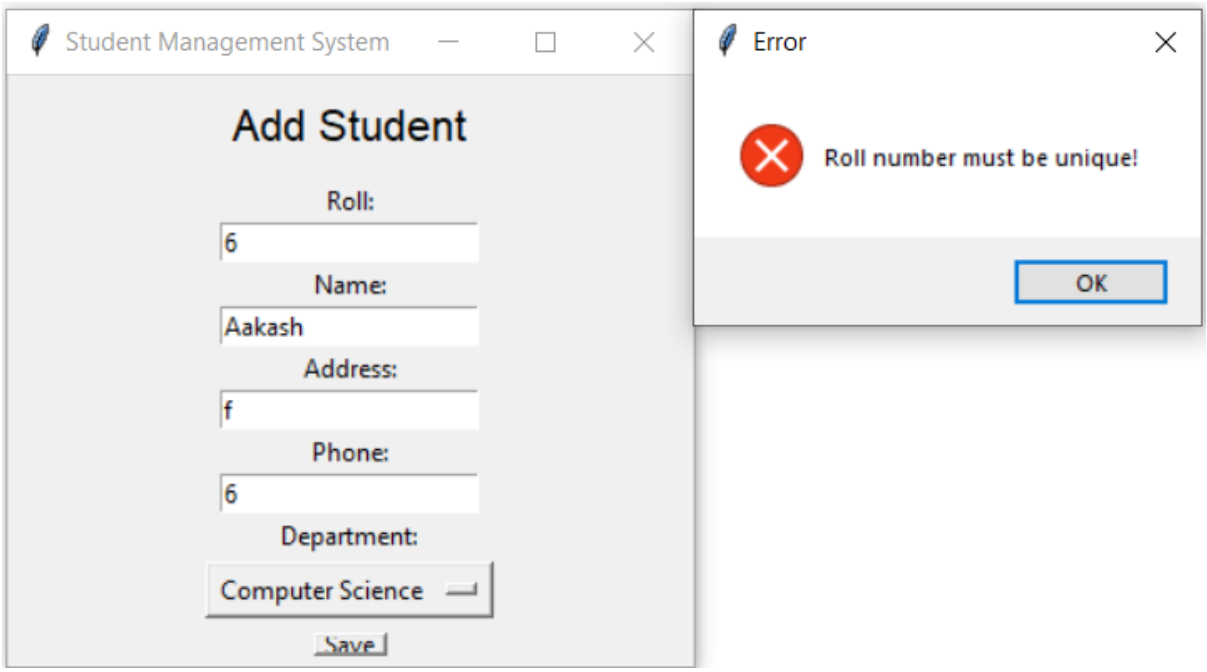
Key Validation:
Ensures UI **does not overlap** when switching between screens.

Output:

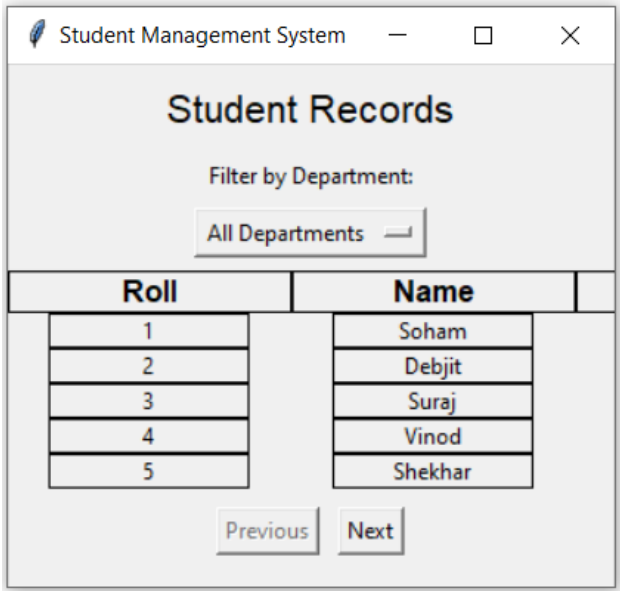
a) Add Student

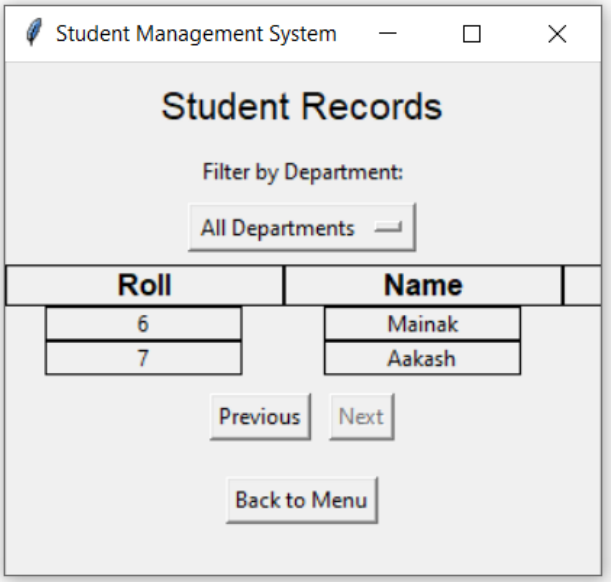


b) Error when Adding Existing Student

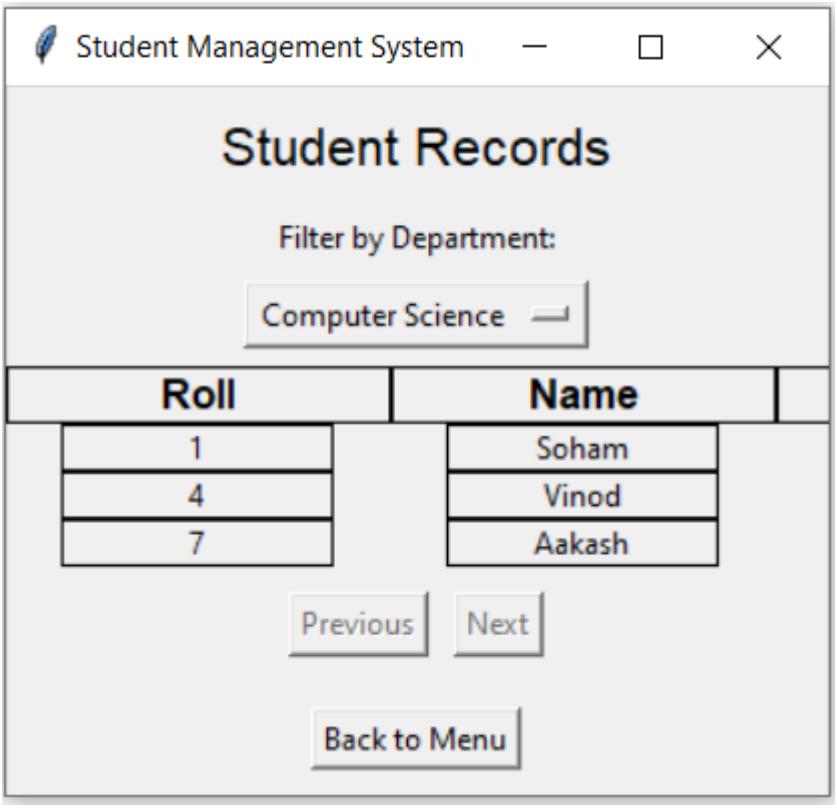


c) Display Students(5 records in 1 page)

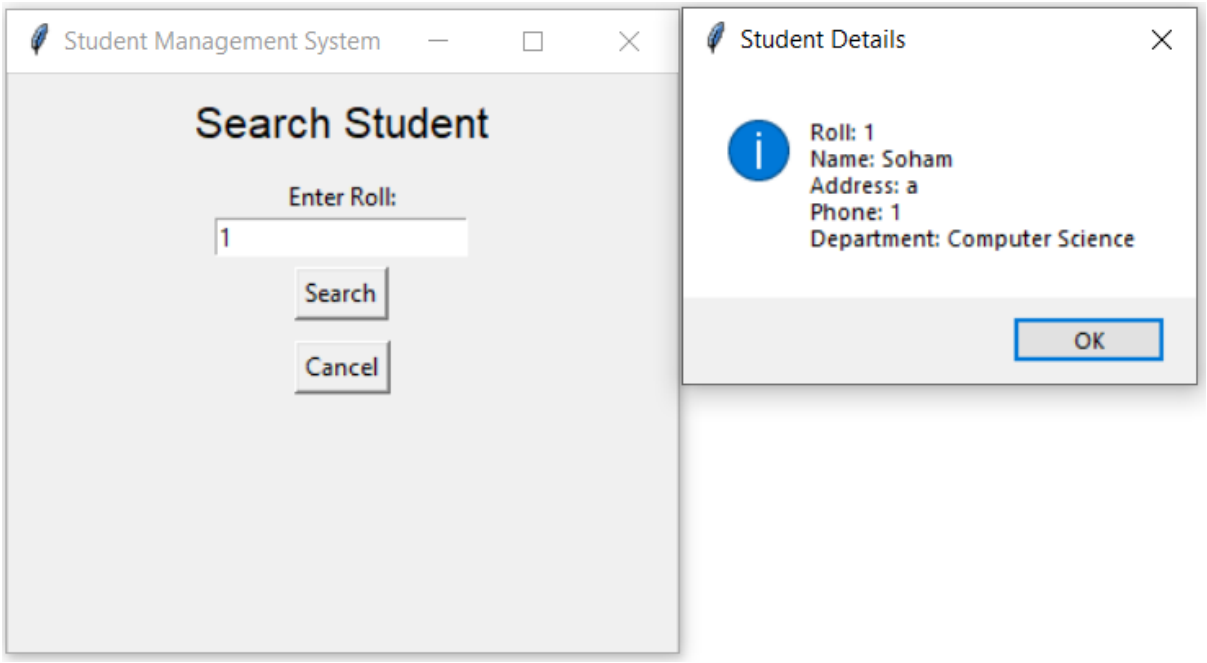




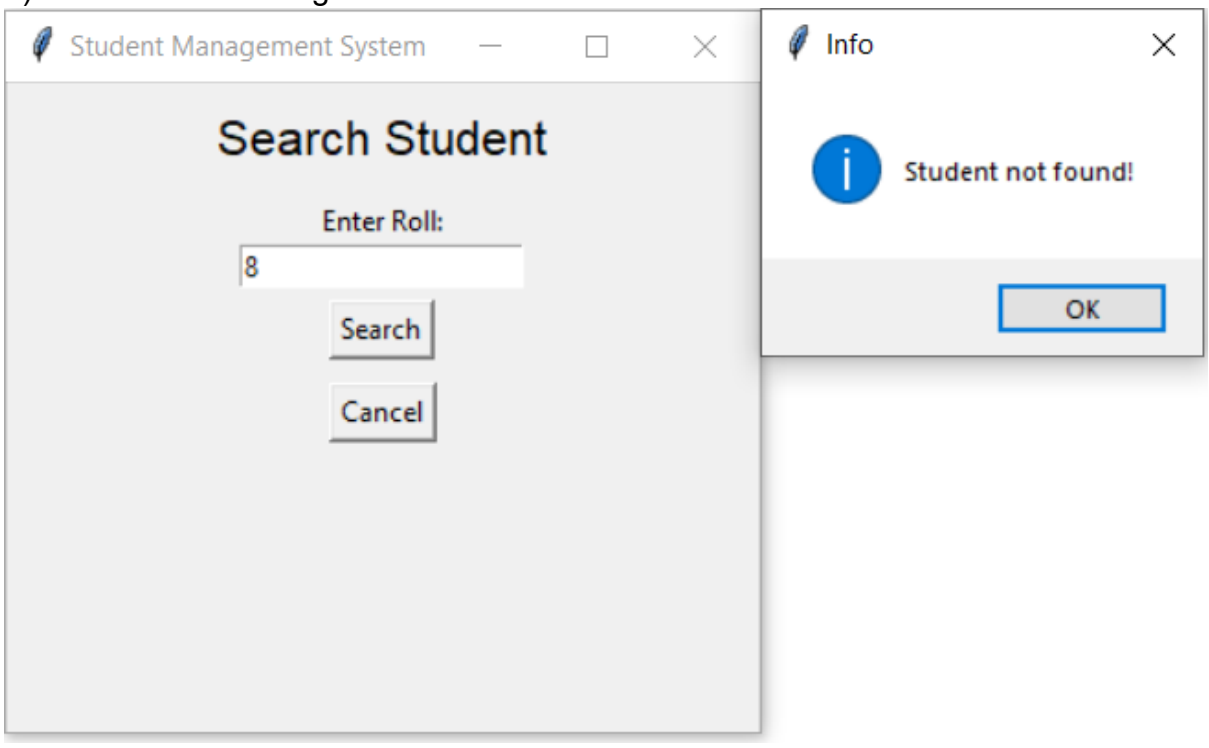
d) Display Students(filter by department)



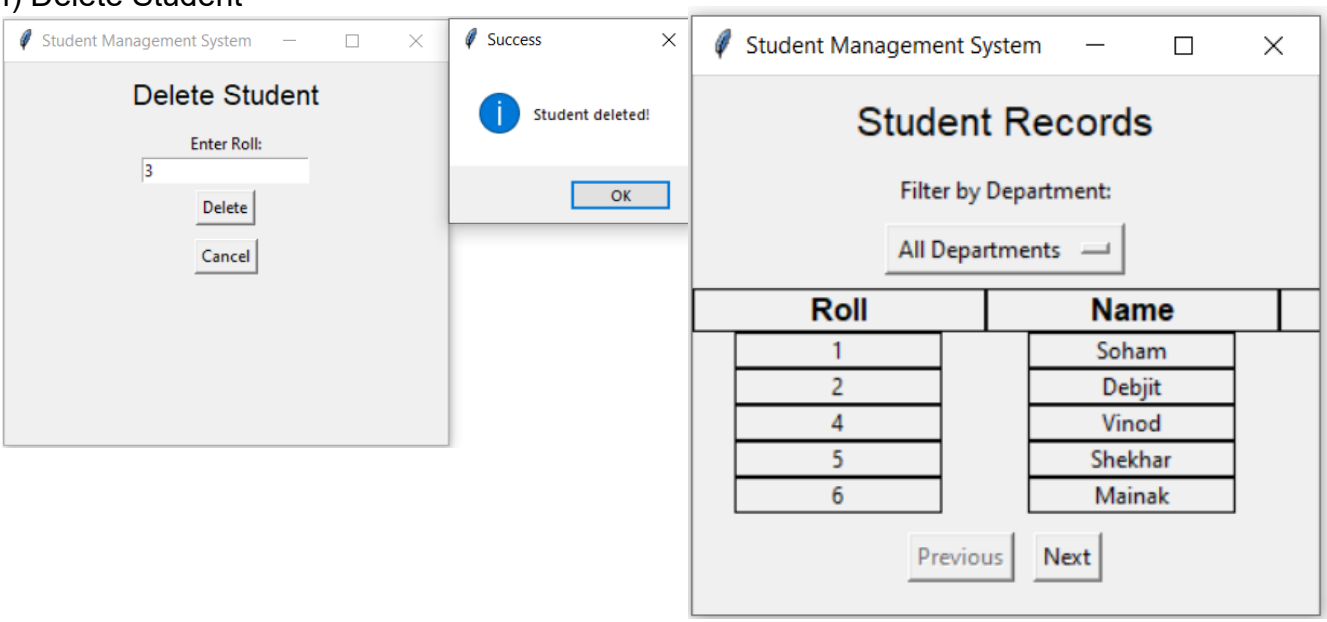
d) Search Student



e) Search Non-existing Student



f) Delete Student



g) Delete non-existing student