

DSA Project Progress Report	
Course Code: 201L DSA	Program: BSCpE
Course Title: Data Structure Analysis	Date Performed: November 08, 2025
Section: 2-B	Date Submitted: November 08, 2025
<b>Name:</b>  Mamano, Kurt Marwin C.  Poliño, Justine  Ramos, Jan Lawrence M.  Uy, Junichiro H.  Vasig, Yuan Hessed O.	<b>Instructor:</b> Engr. Maria Rizette H. Sayo
<b>1. Objectives</b>	
<p>This Project "Money Rider" aims to:</p> <ul style="list-style-type: none"><li>• To add Json File</li><li>• To implement input of income and expenses</li><li>• To use the interactive calendar (months and years)</li><li>• To make the layout of the interface modernize</li><li>• To fix the arrangement of the calendar days and years</li><li>• To implement the log in and log out button</li><li>• To use Json array to store accounts username and password</li><li>• To make a interactable calendar</li><li>• To make an income, and expenses calculation</li></ul>	

## 2. Discussion

The “Money Rider” project successfully met its objectives by effectively applying arrays to achieve efficient data organization and management. The system was developed to support riders in tracking their financial activities, thereby promoting better awareness and control of their income and expenses. Through the proper application of array structures, the project resolved previous inconsistencies in the interactive calendar, ensuring accurate synchronization of dates, months, and years. The modernization of the user interface contributed to a more intuitive and engaging experience, while the integration of login and logout functionalities strengthened data security. Overall, the project demonstrates the practical use of programming concepts in developing a reliable, user-oriented financial management system that aligns technical design with real-world functionality.

## 3. Materials and Equipment

The materials and equipment utilized in the development of the “Money Rider” project include a computer used for coding and testing, Visual Studio Code as the integrated development environment (IDE) for program execution, and GitHub as the collaboration and version control platform. Basic computer peripherals such as a mouse and keyboard were also used to facilitate navigation and input during the coding and testing processes.

- Computer: used to make the source code
- Visual Studio Code: used for running the program
- Github: used to collaborate and manipulate the program
- Mouse: used to navigate the Computer
- Keyboard: Used to type the source code

#### 4. Psedocode

##### **BEGIN PROGRAM**

###### **1. Initialize Variables**

accounts  $\leftarrow$  {}

financial\_data  $\leftarrow$  {}

current\_entries  $\leftarrow$  []

current\_expenses  $\leftarrow$  []

undo\_stack, redo\_stack  $\leftarrow$  []

undo\_expense\_stack, redo\_expense\_stack  $\leftarrow$  []

DATA\_PATH  $\leftarrow$  "money\_rider\_data.json"

###### **2. Call load\_persisted\_state()**

IF data file exists THEN

- Load JSON data into accounts and financial\_data
- ELSE
  - Continue with empty data

##### **FUNCTION: splash\_screen()**

1. Display window **"Money Rider"**

Show two buttons:

[1] Login  $\rightarrow$  call login\_screen()

[2] Create Account  $\rightarrow$  call create\_account\_screen()

2. Wait for user input

##### **FUNCTION: create\_account\_screen()**

1. Prompt:

username  $\leftarrow$  input("Enter username")

password  $\leftarrow$  input("Enter password")

2. IF username  $\neq$  "" AND password  $\neq$  "" THEN

Add account:

accounts[username]  $\leftarrow$  password

- Save data using persist\_state()
- Display "Account Created Successfully"

- Return to splash\_screen()

### 3. ELSE

- Display "Error: Fill all fields"

### **FUNCTION: login\_screen()**

1. Prompt:  
    username  $\leftarrow$  input("Enter username")  
    password  $\leftarrow$  input("Enter password")
2. IF username  $\in$  accounts AND accounts[username] = password THEN
  - Navigate to calendar\_screen()
3. ELSE
  - Display "Error: Wrong Username or Password"

### **FUNCTION: calendar\_screen()**

1. Get current\_year, current\_month from system date
2. Display monthly calendar
3. FOR each day in month DO
  - a. IF day has saved financial data THEN mark as "blue"
  - b. WHEN user clicks on a day:
    - i. IF data exists  $\rightarrow$  call show\_saved\_data(date)
    - ii. ELSE  $\rightarrow$  call income\_screen(day, month, year)
4. Provide **Date Range Calculator**:
  - a. Input start\_date, end\_date
  - b. Initialize total\_income, total\_expenses = 0
  - c. FOR each date in financial\_data:

IF date within range:

total\_income += income

total\_expenses += expenses

- Display range results: income, expenses, and net total
5. Allow navigation "Back" to previous screen

### **FUNCTION: show\_saved\_data(date\_str, day)**

1. Retrieve data  $\leftarrow$  financial\_data[date\_str]
2. Display:
  - a. Total Income
3. Total Expenses  
Net Total = income - expenses

Display detailed lists:

- a. Tab 1: Income entries
- b. Tab 2: Expense entries

4. Options:

- a. **View/Edit** → load entries into memory and call `income_screen()`
- b. **Close** → return to `calendar_screen()`

### **FUNCTION: `income_screen(day, month, year)`**

1. Display current date

WHILE user adds new income entry:

```
name ← input("Customer name")
```

```
amount ← input("Income amount")
```

```
IF valid THEN
```

```
    entry ← (name, amount)
```

```
    Add entry to current_entries
```

```
    Push to undo_stack
```

```
ELSE
```

```
    Display "Error"
```

2. Allow:

- a. **Undo** → remove last entry, move to `redo_stack`

- b. **Redo** → reinsert last undone entry

3. On "Expenses" button:

- a. Save current data via `save_data(date)`

- b. Open `expenses_screen(day, month, year)`

4. On "Back" → `go_back()`

### **FUNCTION: `expenses_screen(day, month, year)`**

1. Display current date

WHILE user adds expense:

```
expense_name ← input("Expense name")
```

```
amount ← input("Expense amount")
```

```
IF valid THEN
```

```
    entry ← (expense_name, amount)
```

```
    Add to current_expenses
```

```
    Push to undo_expense_stack
```

```
ELSE
```

```
    Display "Error"
```

2. Allow:

- a. **Undo / Redo**
- 3. On “Show Totals”:
  - a. Save data → `save_data(date)`
  - b. Go to `total_screen(day, month, year)`
- 4. On “Back” → `go_back()`

### **FUNCTION: `save_data(date_str)`**

Compute:

```
total_income ← sum(income amounts)
total_expenses ← sum(expense amounts)
```

1.

Store data:

```
financial_data[date_str] = {
  income: total_income,
  expenses: total_expenses,
  entries: current_entries,
  expense_entries: current_expenses
}
```

- 2. Call `persist_state()` to save to JSON file.

### **FUNCTION: `total_screen(day, month, year)`**

Compute:

```
total_income ← sum of all current_entries
total_expenses ← sum of all current_expenses
net_total ← total_income - total_expenses
```

1.

Display summary:

```
Total Income: ₱ total_income
Total Expenses: ₱ total_expenses
Net Total: ₱ net_total
```

- 2. Buttons:
  - a. **Back** → return to `expenses_screen()`
  - b. **End** → return to `calendar_screen()`

**FUNCTION: persist\_state()**

1. Convert accounts and financial\_data into serializable JSON format.
2. Write to money\_rider\_data.json.

**FUNCTION: load\_persisted\_state()**

1. IF JSON file exists THEN
  - a. Load its contents.
  - b. Populate accounts and financial\_data.

**PROGRAM END**

1. Exit when all windows are closed.
2. All financial data remains saved persistently.

**End of Pseudo Code****5. Algorithm****1. Initialization**

1. Import necessary modules: tkinter, json, datetime, calendar, etc.
2. Define global variables for:
  - a. Accounts (accounts)
  - b. Income/expense entries and undo/redo stacks
  - c. Financial data storage (financial\_data)
3. Load saved data (money\_rider\_data.json) if it exists.

**2. Splash Screen**

1. Display **Money Rider** title window.
2. Show buttons:
  - a. **Login** → Navigate to Login Screen.
  - b. **Create Account** → Navigate to Account Creation Screen.

**3. Create Account Screen**

1. Ask user for username and password.
2. If both are filled:
  - a. Add to accounts dictionary.
  - b. Save using persist\_state() (write JSON).
  - c. Show success message and return to Splash Screen.
3. Else:

- a. Show error message.

#### 4. Login Screen

1. Ask for username and password.  
If user exists and password matches:
  - a. Open **Calendar Screen**.
2. Else:
  - a. Show error message.

#### 5. Calendar Screen

1. Display the current month's calendar.
2. For each day:
  - a. If financial data exists → mark it in blue.
  - b. If clicked:
    - i. If data exists → show **Saved Data Popup**.
    - ii. Else → go to **Income Screen** for that date.
3. Provide dropdowns for month/year changes.  
Include **Date Range Calculator**:
  - a. Select start and end dates.
  - b. Compute total income, total expenses, and net value within the range.
  - c. Display results in popup.

#### 6. Saved Data Popup

1. Display summary:
  - a. Total income
  - b. Total expenses
  - c. Net total
2. Tabs for detailed entries (Income / Expense).
3. Buttons:
  - a. **View/Edit** → Opens Income Screen with existing data.
  - b. **Close** → Closes popup.

#### 7. Income Screen

1. Display selected date.
2. Inputs:
  - a. Customer name
  - b. Income amount
3. Buttons:
  - a. **Enter** → Adds entry to list and undo stack.
  - b. **Undo / Redo** → Manages entry changes.
  - c. **Expenses** → Saves current data and opens Expenses Screen.
  - d. **Back** → Returns to previous screen.

## 8. Expenses Screen

1. Display selected date.
2. Button **Add Option** → Opens popup for adding expense item.
3. On saving:
  - a. Add item to list and undo stack.
4. Buttons:
  - a. **Undo / Redo**
  - b. **Show Totals** → Save data and navigate to Total Screen.
  - c. **Back** → Return to previous screen.

## 9. Save Data Function

1. Calculate total income and total expenses.
2. Save all entries in `financial_data[date_str]`.
3. Write updated JSON file using `persist_state()`.

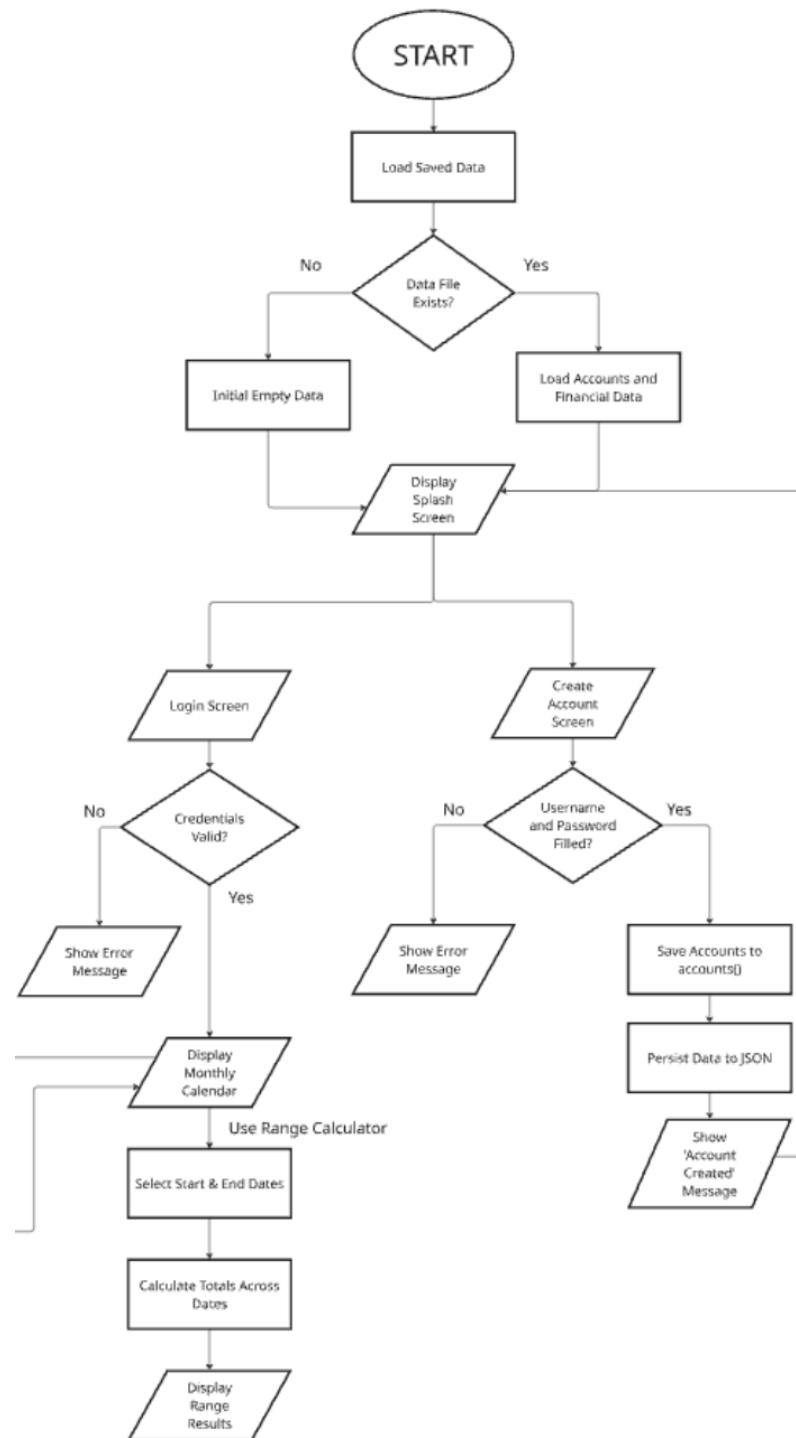
## 10. Total Screen

1. Show:
  - a. Total income
  - b. Total expenses
  - c. Net balance (green for positive, red for negative).
2. Buttons:
  - a. **Back** → Go back to Expenses Screen.
  - b. **End** → Return to Calendar Screen.

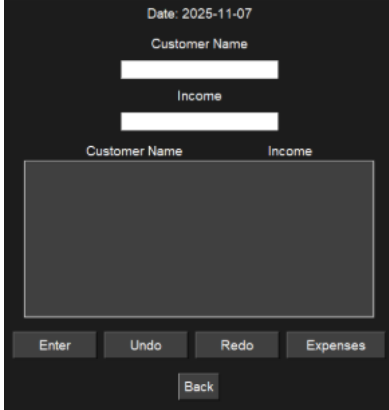
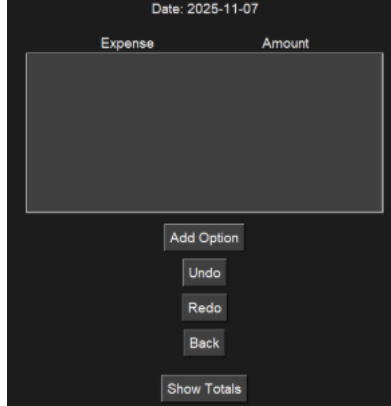
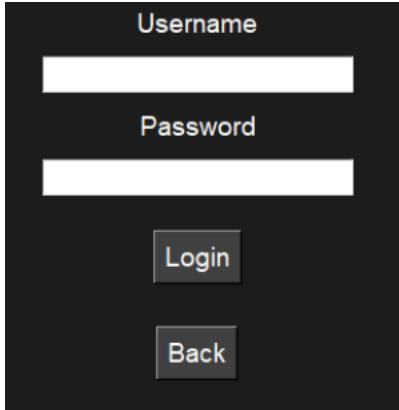
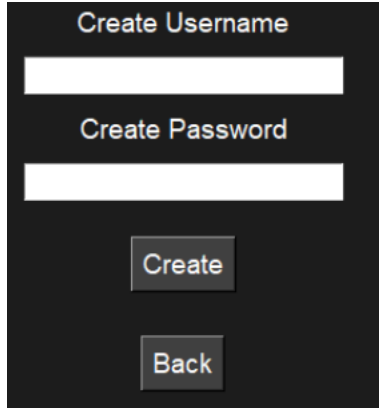
## 11. Exit Program

1. When user closes the main window, all data remains saved persistently.

## 6. Flowchart



## 7. Output

Income Screen		Expenses Screen	
 The Income Screen UI mockup features a dark background. At the top, it displays 'Date: 2025-11-07'. Below this are two input fields: 'Customer Name' and 'Income'. A large, empty rectangular area is positioned below the input fields. At the bottom, there are four buttons: 'Enter', 'Undo', 'Redo', and 'Expenses', with a 'Back' button centered below them.		 The Expenses Screen UI mockup has a dark background. It shows 'Date: 2025-11-07' at the top. Below are two columns: 'Expense' and 'Amount'. A large, empty rectangular area is below these columns. At the bottom, there are five buttons: 'Add Option', 'Undo', 'Redo', 'Back', and 'Show Totals'.	
Login Page		Sign Up Page	
 The Login Page UI mockup has a dark background. It features two input fields: 'Username' and 'Password'. Below the input fields are two buttons: 'Login' and 'Back'.		 The Sign Up Page UI mockup has a dark background. It features two input fields: 'Create Username' and 'Create Password'. Below the input fields are two buttons: 'Create' and 'Back'.	
Calendar Grid		Result Page	

November

2025

Mon	Tue	Wed	Thu	Fri	Sat	Sun
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Date Range Calculator

From: 1

November

2025

To: 1

November

2025

Calculate Range

Back

Financial Summary

Date: 2025-11-06

Total Income:

P845.00

Total Expenses:

P256.00

Net Total:

P589.00

Back

End

## 8. Procedure

Of course in order to be able to use an application, we must first be able to make accounts and have some privacy from other users, we must be able to make an account. So we made a sign up and sign in page, and when you sign up, the code makes a json file inside the folder the main code is contained in, the username and password is stored there:

```
DATA_PATH = Path(__file__).with_name("money_rider_data.json")

def load_persisted_state():
    global accounts, financial_data
    if not DATA_PATH.exists():
        return
    try:
        with DATA_PATH.open("r", encoding="utf-8") as data_file:
            raw = json.load(data_file)
    except (json.JSONDecodeError, OSError):
        return

    accounts = raw.get("accounts", {})

def persist_state():
    serializable_data = {
        "accounts": accounts,
        "financial_data": { ... } # existing serialization for financial_data
    }
    try:
        with DATA_PATH.open("w", encoding="utf-8") as data_file:
```

```

        json.dump(serializable_data, data_file, indent=2)
    except OSError:
        messagebox.showerror("Error", "Failed to save data to disk.")
# ...existing code...

def create_account_screen():
    create = tk.Tk()
    create.title("Create Account")
    create.geometry("570x700")
    create.configure(bg="#1C1C1C")

    tk.Label(create, text="Create Username", bg="#1C1C1C", fg="white",
font=("Bubblegum Sans", 14)).pack(pady=5)
    username_entry = tk.Entry(create, font=("Bubblegum Sans", 14))
    username_entry.pack(pady=5)

    tk.Label(create, text="Create Password", bg="#1C1C1C", fg="white",
font=("Bubblegum Sans", 14)).pack(pady=5)
    password_entry = tk.Entry(create, show="*", font=("Bubblegum Sans", 14))
    password_entry.pack(pady=5)

    def create_account():
        username = username_entry.get()
        password = password_entry.get()
        if username and password:
            accounts[username] = password
            persist_state()
            messagebox.showinfo("Success", "Account Created!")
            go_back(create)
        else:
            messagebox.showerror("Error", "Fill all fields")

    tk.Button(create, text="Create", font=("Bubblegum Sans", 14),
bg="#404040", fg="white", command=create_account).pack(pady=20)
    tk.Button(create, text="Back", font=("Bubblegum Sans", 14), bg="#404040",
fg="white",
        command=lambda: go_back(create)).pack(pady=10)

def login_screen():
    login = tk.Tk()
    login.title("Login")
    login.geometry("570x700")
    login.configure(bg="#1C1C1C")

    tk.Label(login, text="Username", bg="#1C1C1C", fg="white",
font=("Bubblegum Sans", 14)).pack(pady=5)

```

```

username_entry = tk.Entry(login, font=("Bubblegum Sans", 14))
username_entry.pack(pady=5)

tk.Label(login, text="Password", bg="#1C1C1C", fg="white",
font=("Bubblegum Sans", 14)).pack(pady=5)
password_entry = tk.Entry(login, show="*", font=("Bubblegum Sans", 14))
password_entry.pack(pady=5)

def validate_login():
    username = username_entry.get()
    password = password_entry.get()
    if username in accounts and accounts[username] == password:
        login.destroy()
        navigate_to("calendar", calendar_screen)
    else:
        messagebox.showerror("Error", "Wrong Username or Password!")

tk.Button(login, text="Login", font=("Bubblegum Sans", 14), bg="#404040",
fg="white", command=validate_login).pack(pady=20)
tk.Button(login, text="Back", font=("Bubblegum Sans", 14), bg="#404040",
fg="white",
        command=lambda: go_back(login)).pack(pady=10)
# ...existing code...

```

We also added a go back function, where if you clicked it, you can go back to the previous page you just visited. It made use of doubly linked list, using .next and .prev:

```

def go_back(window):
    global current_page
    if current_page and current_page.prev:
        prev_node = current_page.prev
        prev_node.next = None
        current_page = prev_node
        window.destroy()
        prev_node.render_fn(*prev_node.args)
    else:
        window.destroy()

```

We also made use of stack data struct to make a undo and redo button. We used this in Income screen and expense screen so we can undo the inputs we made and also redo them

Undo and Redo function in Income screen:

```

def undo():
    if listbox.curselection():
        index = listbox.curselection()[0]
        redo_stack.append(current_entries.pop(index))

```

```

        listbox.delete(index)
    elif current_entries:
        redo_stack.append(current_entries.pop(0))
        listbox.delete(0)

def redo():
    if redo_stack:
        entry = redo_stack.pop()
        current_entries.insert(0, entry)
        listbox.insert(0, format_row(entry[0], entry[1]))

```

Undo and Redo function in Income screen:

```

def undo():
    if listbox.curselection():
        index = listbox.curselection()[0]
        redo_expense_stack.append(current_expenses.pop(index))
        listbox.delete(index)
    elif current_expenses:
        redo_expense_stack.append(current_expenses.pop(0))
        listbox.delete(0)

def redo():
    if redo_expense_stack:
        entry = redo_expense_stack.pop()
        current_expenses.insert(0, entry)
        listbox.insert(0, format_row(entry[0], entry[1]))

```

We made the interface in a form of Calendar Grid, the libraries we used are tkinter for the gui, calendar library, pathlib, json. The code is too long, that's why I'll only send a link from collab containing all the source code.

Link:

[https://colab.research.google.com/drive/12gXE4hfPKNEjJPAAIG34htT9mvS1634K#scrollTo=tLBjWQ\\_Zux5G&line=536&uniqifier=1](https://colab.research.google.com/drive/12gXE4hfPKNEjJPAAIG34htT9mvS1634K#scrollTo=tLBjWQ_Zux5G&line=536&uniqifier=1)

## 9. Conclusion

In conclusion, the “Money Rider” project successfully achieved its primary goal of creating a functional and user-friendly financial management system designed to assist riders in organizing and monitoring their daily income and expenses. By integrating programming concepts such as arrays and JSON, the project demonstrated effective data handling, reliable storage, and accurate financial tracking within an interactive and modernized interface. The inclusion of features such as Undo and Redo functions, secure login and logout options, and an interactive calendar enhanced both usability and data security. Overall, the project highlights the importance of combining technical programming knowledge with practical application to develop solutions that address real-world needs. The “Money Rider” system serves as a valuable tool for financial organization while showcasing the team’s technical competence and innovative approach to software development.