Even Semester (2019)



**BINUS UNIVERSITY**



**BINUS INTERNATIONAL**



**Assignment Cover Letter**

**(Group Work)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | |  |  |  |  |
| **Student Information**: | **Surname** | | |  |  | **Given Names**  **David**  **Millen** | **Student ID Number**  **2201797304  2201797531** |
| 1.  2. | **Amadeo Dwiputra** | |
|  |  |  |  |  |  |  |  |
| **Course Code** | **: COMP6571** |  |  |  |  | **Course Name** | **: Data Structure and Algorithms** |
| **Class** | **: L2AC** |  |  |  |  | **Name of Lecturer(s)** | : **Kartiko Eko Putranto** |
|  |  |  |  |  |  |  |  |
| **Major** | **: CS** |  |  |  |  |  |  |
| **Title of Assignment**  (if any) | : Smart Financial Manager |  |  |  |  |  |  |
| **Type of Assignment**    **Submission Pattern** | **: Final Project** | |  |  |  |  |  |
| **Due Date** | **: 29 – 07 – 2019** | |  |  |  | **Submission Date** | **: 01 – 07 – 2019** |

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer’s instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

# Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

# Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: (Name of Student)

**David Amadeo & Millen Dwiputra**

**II. Table of Contents**

1. **Cover**
2. **Table of content**
3. **Problem description**
4. **Solution design**
5. **Alternative data structure and theoretical analysis**
6. **Programme manual & execution outcome**
7. **How code works**
8. **Link to GIT website**

**III. Project description**

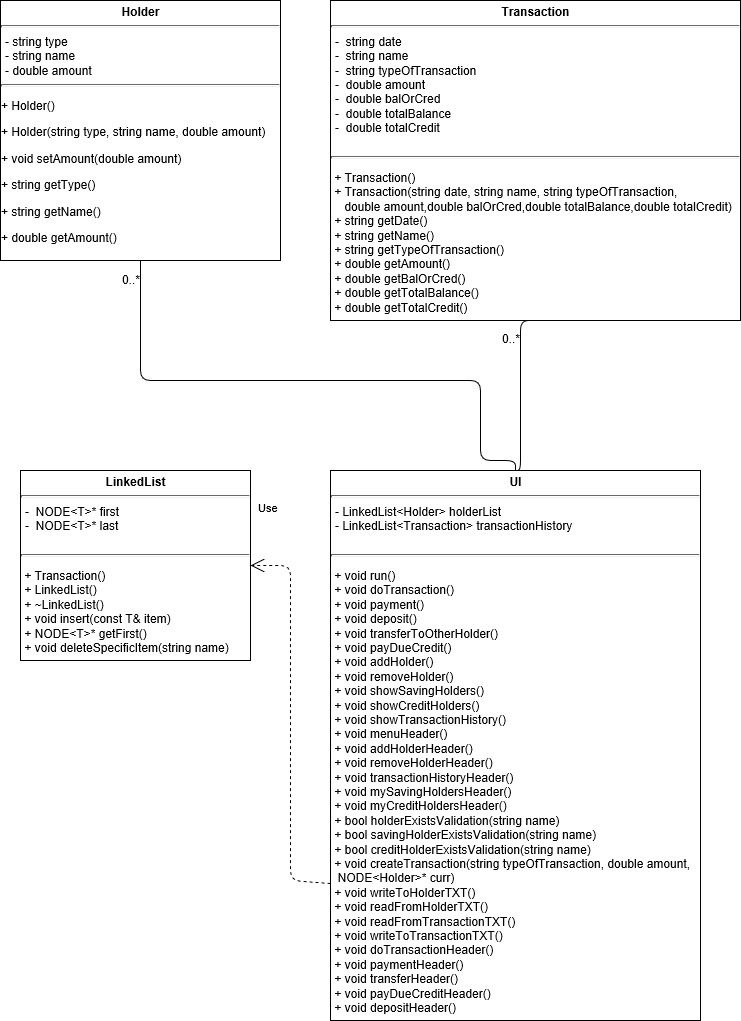
Humans are social beings, we interact with one another everyday and it has been our way of life. We depend on another for terms of service, every human being in the world have their own role in the society and we need each other to maintain a healthy balance. Even before a universal currency have been established, we humans have found a way to exchange services with another. Everyone uses it, to pay for various kind of activities that requires someone else’s services.

Up until the 21st century, we have maintained this relationship between each other and with the invention of a universal currency system in the world, exchanging in services with other people have never been easier. Especially in this day and age, without this universal currency, money, we are unable to live. We are required to have money to do mostly everything. It sounds all great and good that we can use other people’s service in exchange with money however not everything is good and well.

An average person will have to spend for different kinds of things in their daily lives for example: rent, electricity bills, taxes, food and others. When there are so many kinds of expenses, it is extremely difficult to keep up on where your money goes. To keep up how much money you are earning, how much debt you have, whether you will be able to pay for daily necessities and many more. Moreover, it will be highly confusing to know how much money one man has.

Even dating back to the old days when people started making banks so that people can store their money safely to a trusted holder and lately with the latest technological advancement of financial technology, the creation of fin-techs such as: OVO, Dana, GoPay, makes it difficult to keep track of where your money is. Specially to know how much money you may have when your money is scattered in different money holders.

**IV. Solution design**



**V. Alternative data structure and theoretical analysis**

The data structure used in the programme, Smart Financial Manager, is a single linked list and an abstract data type. But before we get into alternative data structure that could similarly solve the problem the Smart Financial Manager could do, let’s explain why we chose to use a single linked list and an abstract data type for our programme.

We chose to use a single linked list because it is a dynamic data structure. The user will be able to add any amount of data without any difficulty. Moreover, the single linked list won’t be burdening the memory too much and will use the memory efficiently. Unlike stack or queue, the linked list can add a new node through the end or beginning of the linked list and when a node wants to be deleted, it does not follow a certain pattern rather it can choose directly which node that would be deleted and delete it.

We also chose to use an abstract data type for our linked list because it enables us to modify the single linked list in a way that we can insert any data type (of the same type) in the linked list. So, we can add an object, string, integer and many more just by initiating it when instantiating the class to become an object.

Some other alternative data structure could be, a static array. In theory, swapping a single linked list for a static array would not be such a terrible idea. It performs very similarly to a single linked list. A static array would be able to store data, we could add or delete an element to array. So, in theory it would work however we opted to use a single linked list instead of a static array because with a static array, we would need to pre-determine, in code, the size of the array. When someone reaches the limit, they would not be able to add anymore elements into the array.

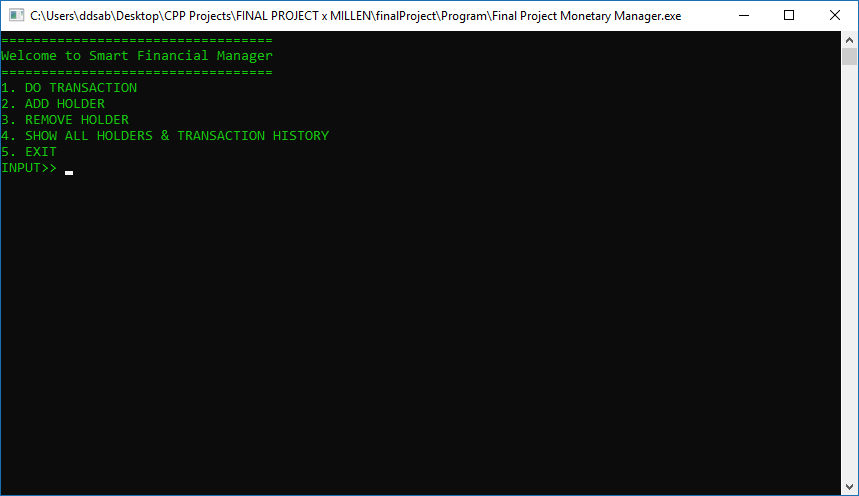
All in all, we believe that the single linked list will be the most suitable and most customizable data structure to use in our programme. Furthermore, the use of the abstract data will create the programme more efficient so that we are able to make different single linked list data structure for different data types with just one line of code.

**VI. Programme manual and execution outcome**

The Smart Financial Manager is a programme text-based application that enables users to keep track of all their money. To further discuss what the Smart Financial Manager can do, below is the list of the features in the Smart Financial Manager:

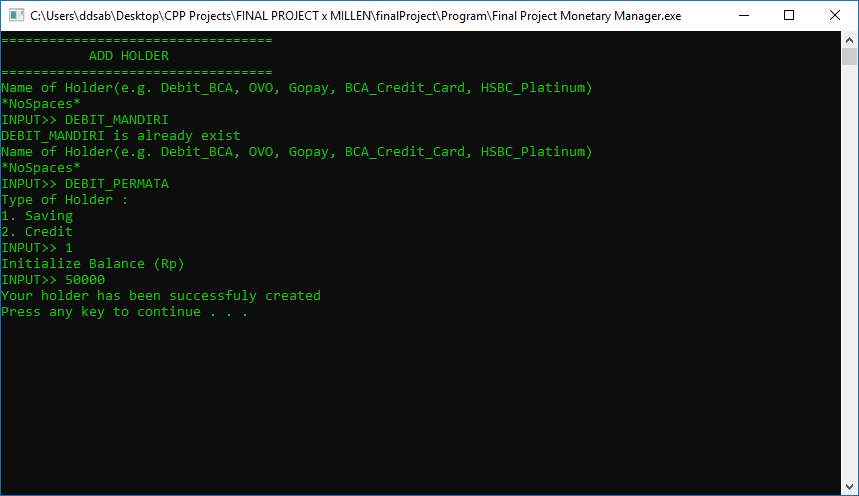
1. Add a holder
   1. Debit
   2. Credit
2. Remove a holder
3. Do a transaction
   1. Make a payment
   2. Deposit money
   3. Transfer
   4. Pay due credit
4. Show all holders and transaction history
5. Exit

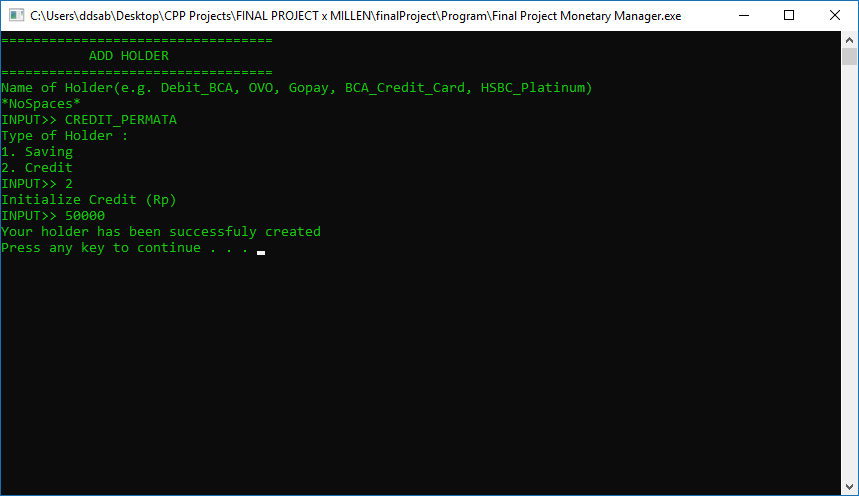
When the user starts the Smart Financial Manager on their PC, they will be directed to a window where there is a menu to what they want to do.

  
 1.1 Smart Financial Manager Main Menu

**I. Add a holder**

When the user wants to add a holder, they press menu 2.

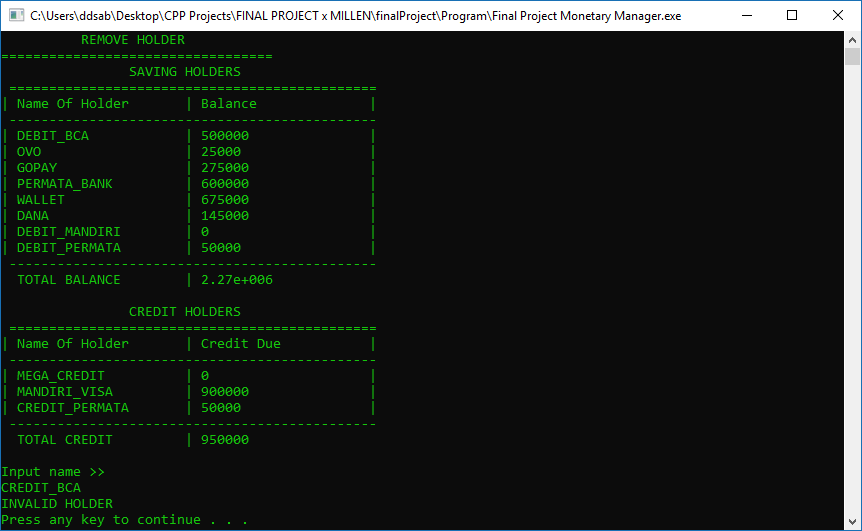
  
 1.2 Smart Financial Manager add holder feature saving example

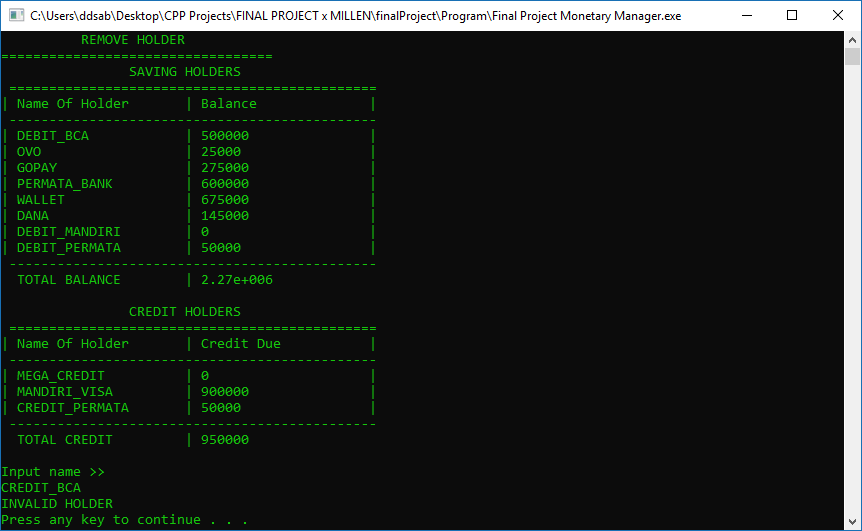
  
 1.3 Smart Financial Manager add holder feature credit example

The user will be asked to input the name of the holder, determine whether it is a saving or credit type of money holder, then input the initial balance of the money holder. Note, it is also visible in the screenshot the validation process of adding a holder. The user can not add a holder with the same name.

**II. Remove holder**

When the user wants to delete a holder, they press menu 3.

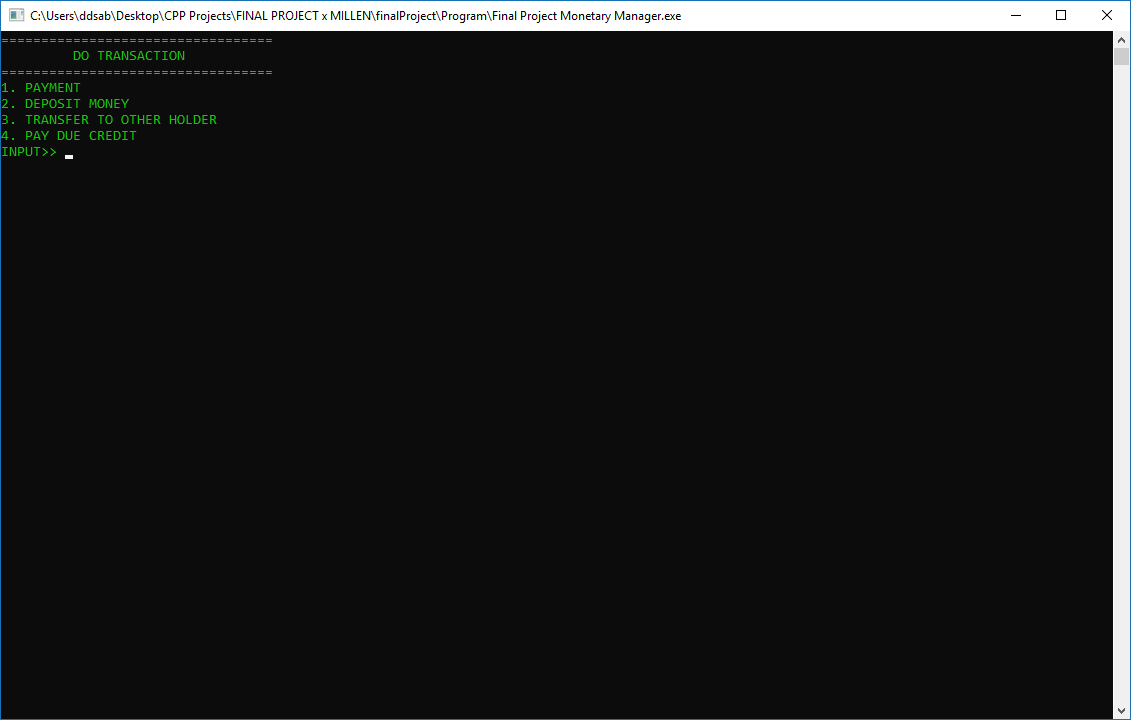
  
 1.4 Smart Financial Manager delete holder feature validation example

  
 1.5 Smart Financial Manager delete holder feature example

The user will be provided with the list of their different holders that they have inputted prior. Then they will be asked to input the name of the holder that they would like to delete. Note, there is also the example of validation in the delete holder process. The user must input a name that is present in the holder list, otherwise the process will be evoked.

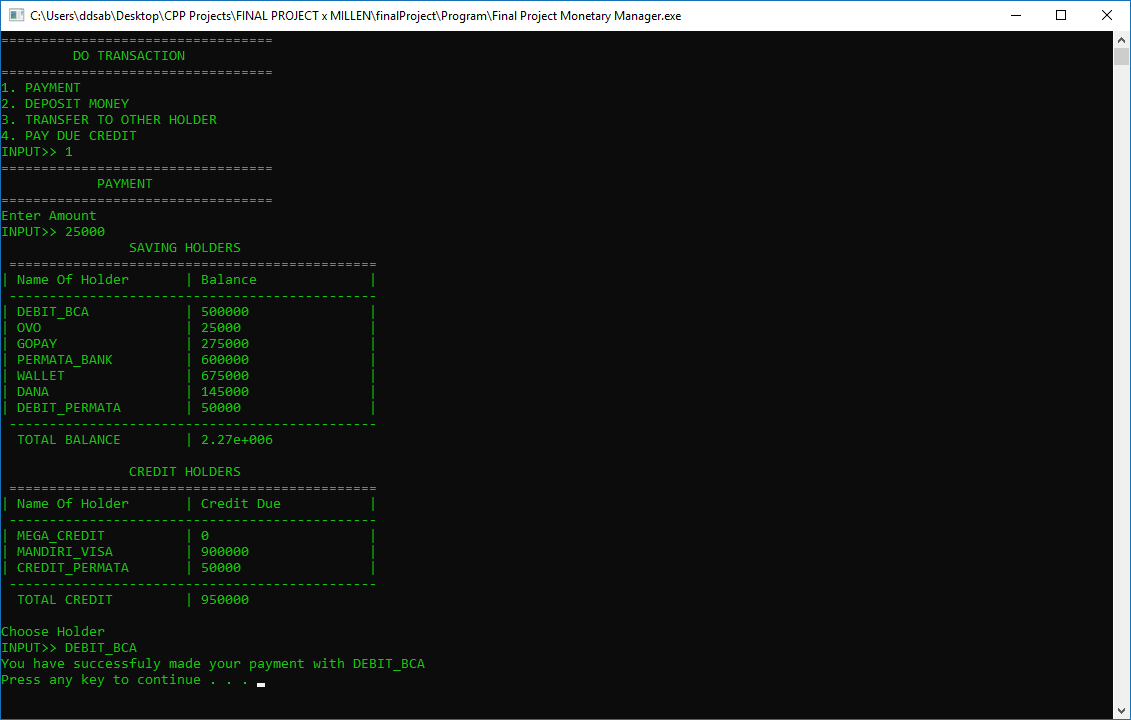
**III. Do a transaction**

When the user wants to do a transaction, they press menu 1.

  
 1.6 Smart Financial Manager do transaction feature menu

**i. Make a payment**

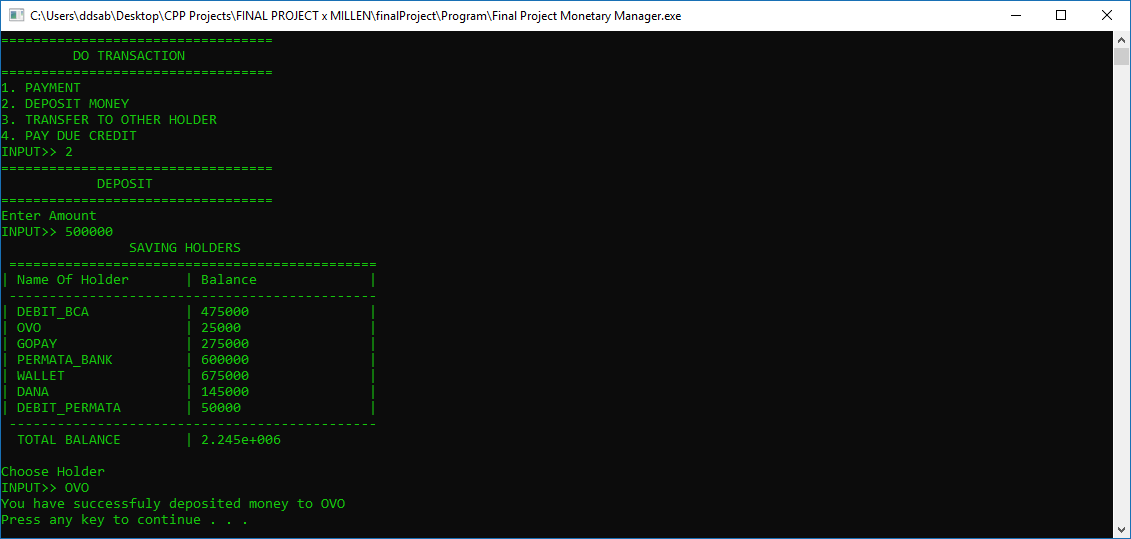
When the user wants to make a payment, they press menu 1.

  
 1.7 Smart Financial Manager making a payment

When the user wants to complete a payment, the user will be asked to input the amount that they would like to pay, then choose the holder that they would like to complete the transaction with, then the process is complete.

**ii. Deposit money**

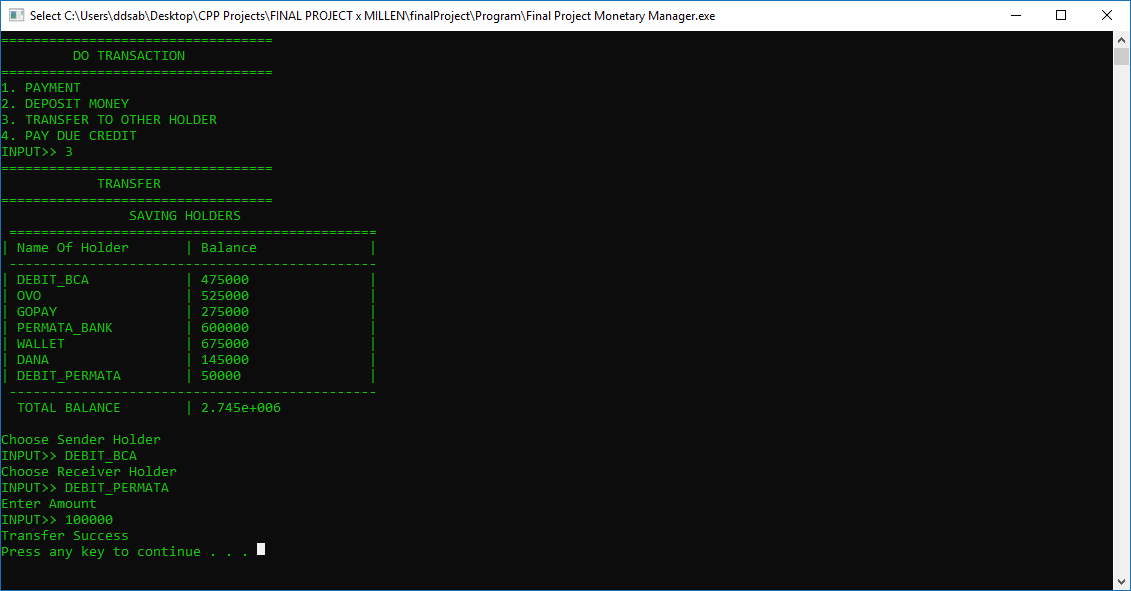
When the user wants to deposit money, they press menu 2.

  
 1.8 Smart Financial Manager deposit money

When the user wants to deposit money, first they must input the amount of money they would like to input, then they choose which holder they would like to deposit the money to.

**iii. Transfer to another holder**

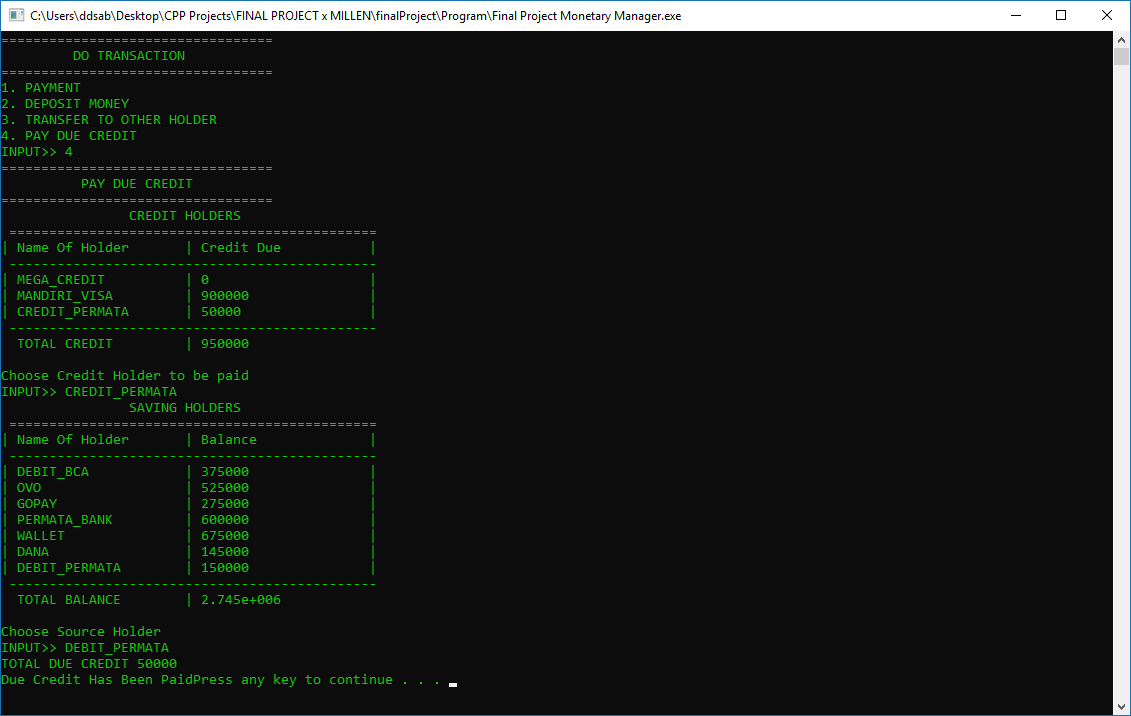
When the user wants to transfer money from one holder to another, they press menu 3.

  
 1.9 Smart Financial Manager transfer to another holder

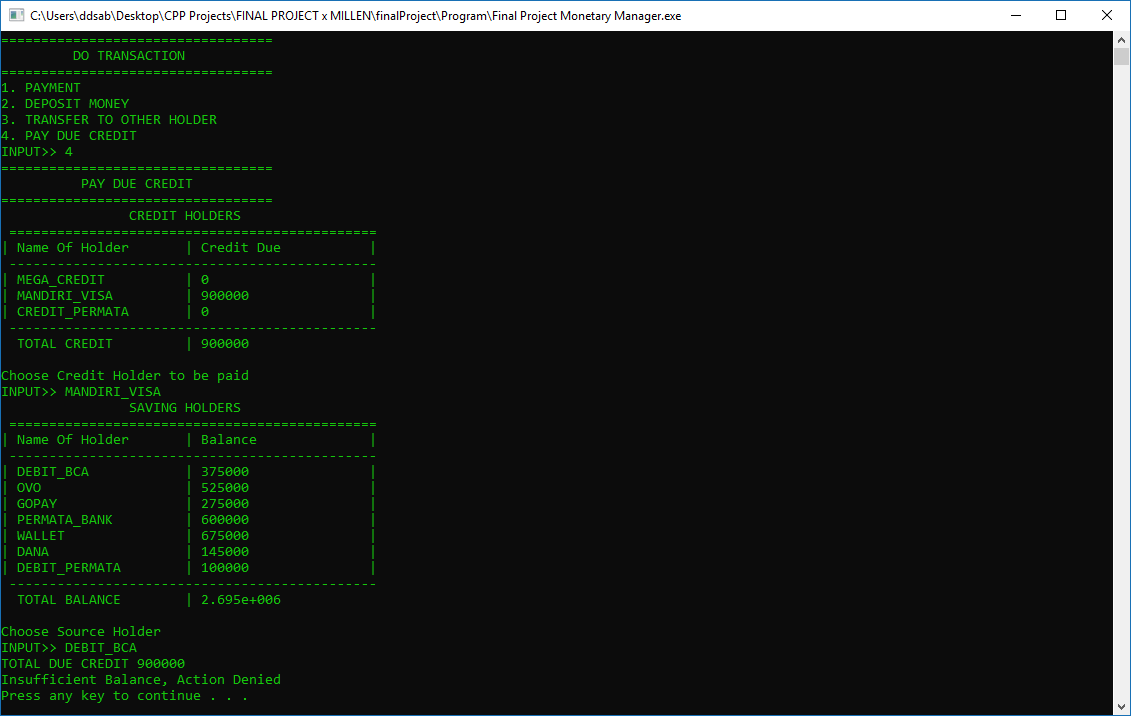
When the user wants to transfer money from one holder to the other, they first input the sender holder, then the destination holder and the amount of money they would like to transfer. This is useful for topping up fin-tech holder balance such as OVO or GoPay.

**iv. Pay due credit**

When the user wants to pay due credit, they press menu 4.

  
 1.10 Smart Financial Manager paying due credit

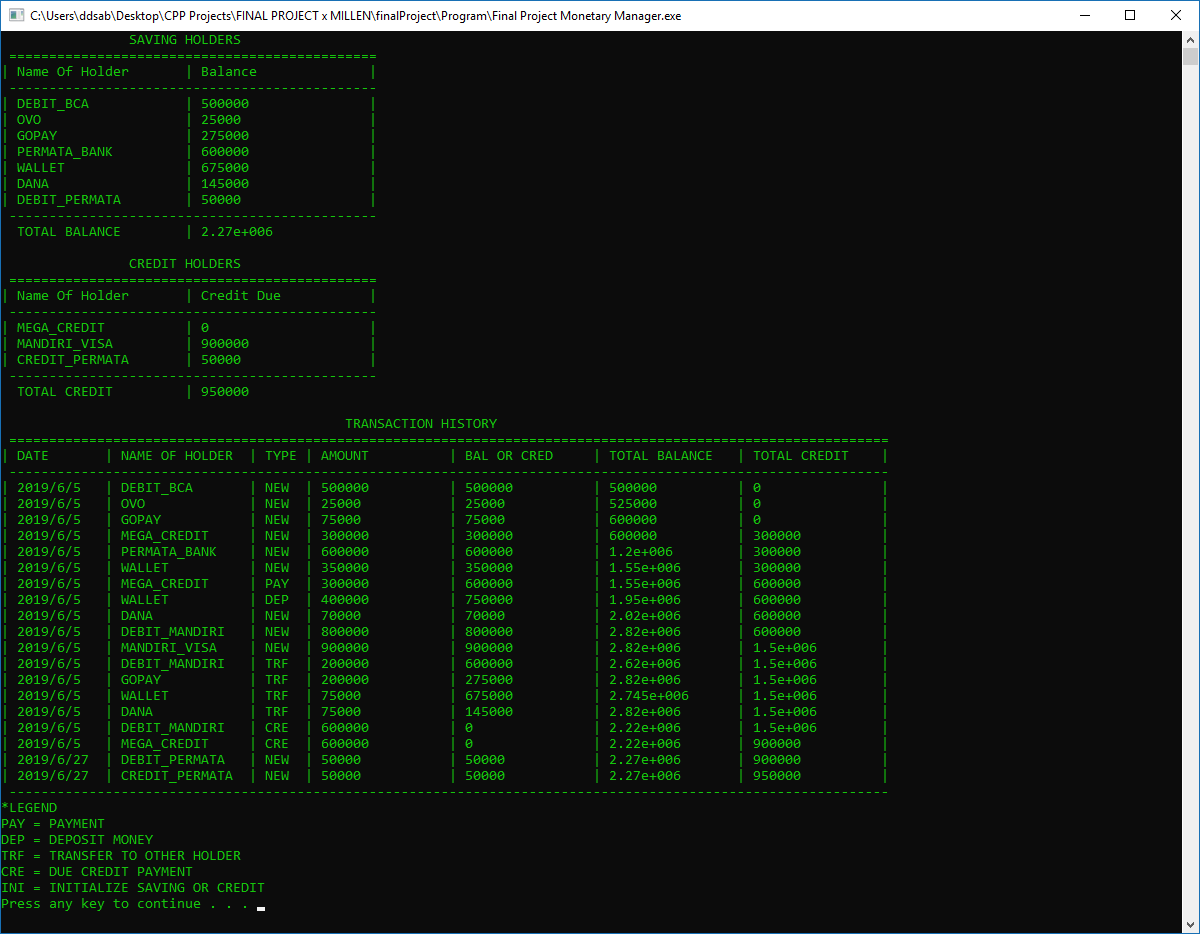
When the user wants to pay off a due credit card debt, they first choose which credit card they would like to pay off. Then they choose the saving holder that they would like to pay the credit card due with.

  
 1.11 Smart Financial Manager paying due credit validation

For example, if the credit due amount exceeds the balance of the saving holder that the user wants to pay with, the action will be evoked. So, the user has to make sure that the saving holder that they would like to use to pay the credit due has sufficient balance to complete the process.

**IV. Show all holders & transaction history**

When the user wants to see all the holders that they have inputted and their transaction history, they press menu 4.

1.12 Smart Financial Manager show all holders & transaction history

In the window, the list of all the saving holders, credit holders and all their transaction history will be printed out along with the legend. For the holders, the holder name and balance will be printed out along with their total balance and credit while for the transaction history, the date, name of holder, type, amount, value of balance or credit, the total balance and the total credit will be print out on a neat table, easy to read and easy to understand.

**V. Exit**

When the user wants to exit the application, press the menu 5.

**VI. How code work**

To truly understand how the programme works, let’s divide the explanation into the features in Smart Financial Manager

1. Add a holder

void UI::addHolder()

{

    addHolderHeader();

    string name;

    string type;

    double amount;

    int num = 0;

    //CHECKS WHETHER IT IS UNIQUE (DOES NOT EXIST IN THE HOLDER LIST)

    do

    {

        if (num != 0)

        {

            cout << name << " already exists" << endl;

        }

        cout << "Name of Holder(e.g. Debit\_BCA, OVO, GOPAY, BCA\_CREDIT, HSBC\_Platinum, WALLET)\n"

                "\*NoSpaces\*\n"

                "INPUT>> ";

        cin >> name;

        num += 1;

        transform(name.begin(), name.end(), name.begin(), ::toupper);

    } while (holderExistsValidation(name));

    //ASKS FOR THE TYPE OF HOLDER

    cout << "Type of Holder :\n"

            "1. Saving\n"

            "2. Credit\n"

            "INPUT>> ";

    cin >> type;

    //INSERT HOLDER DEPENDING ON THE TYPE

    if (type == "1")

    {

        cout << "Initialize Balance (Rp) \n"

                "INPUT>> ";

        cin >> amount;

        Holder holder("S", name, amount);

        holderList.insert(holder);

        if (amount != 0)

        {

            NODE<Holder> \*curr = holderList.getFirst();

            do

            {

                if (curr->link == NULL)

                {

                    createTransaction("NEW", amount, curr);

                }

                curr = curr->link;

            } while (curr != NULL);

        }

    }

    if (type == "2")

    {

        cout << "Initialize Credit (Rp) \n"

                "INPUT>> ";

        cin >> amount;

        Holder holder("C", name, amount);

        holderList.insert(holder);

        if (amount != 0)

        {

            NODE<Holder> \*curr = holderList.getFirst();

            do

            {

                if (curr->link == NULL)

                {

                    createTransaction("NEW", amount, curr);

                }

                curr = curr->link;

            } while (curr != NULL);

        }

    }

    cout << "Your holder has been successfuly created" << endl;

}

1. Remove a holder

void UI::removeHolder()

{

    removeHolderHeader();

    showSavingHolders();

    showCreditHolders();

    string name;

    cout << "Input name >> " << endl;

    cin >> name;

    transform(name.begin(), name.end(), name.begin(), ::toupper);

    holderList.deleteSpecificItem(name);

}

1. Do a transaction
   1. Make a payment

void UI::payment()

{

    paymentHeader();

    double amount;

    string name;

    //Enter Amount

    cout << "Enter Amount\n"

            "INPUT>> ";

    cin >> amount;

    NODE<Holder> \*curr = holderList.getFirst();

    if (curr == NULL)

    {

        cout << "You do not have a holder, Please create a new one" << endl;

    }

    else

    {

        //CHOOSES HOLDER WITH VALIDATION

        showSavingHolders();

        showCreditHolders();

        int num = 0;

        do

        {

            if (num != 0)

            {

                cout << "INVALID HOLDER\n";

            }

            cout << "Choose Holder\n"

                    "INPUT>> ";

            num += 1;

            cin >> name;

            transform(name.begin(), name.end(), name.begin(), ::toupper);

        } while (!holderExistsValidation(name));

        //  CHANGES THE VALUE TO THE NEW ONE

        NODE<Holder> \*curr = holderList.getFirst();

        while (curr != NULL)

        {

            if (name == curr->data.getName() && curr->data.getType() == "S")

            {

                if (curr->data.getAmount() >= amount)

                {

                    double finalAmount = curr->data.getAmount() - amount;

                    curr->data.setAmount(finalAmount);

                    //                  WRITE TRANSACTION TO transactionHistory

                    //                  -------------------------------------------------------------------------------------------------

                    createTransaction("PAY", amount, curr);

                    //                  -------------------------------------------------------------------------------------------------

                    cout << "You have successfuly made your payment with " << name << endl;

                }

                else

                {

                    cout << "Insufficient Balance, Action Denied" << endl;

                }

                break;

            }

            else if (name == curr->data.getName() && curr->data.getType() == "C")

            {

                double finalAmount = curr->data.getAmount() + amount;

                curr->data.setAmount(finalAmount);

                cout << "You have successfuly made your payment with " << name << endl;

                //              WRITE TRANSACTION TO transactionHistory

                //              -------------------------------------------------------------------------------------------------

                createTransaction("PAY", amount, curr);

                //              -------------------------------------------------------------------------------------------------

                break;

            }

            curr = curr->link;

        }

    }

}

* 1. Deposit money

void UI::deposit()

{

    depositHeader();

    NODE<Holder> \*saving = holderList.getFirst();

    while (saving != NULL)

    {

        if (saving->data.getType() == "S")

        {

            break;

        }

        saving = saving->link;

    }

    if (saving == NULL)

    {

        cout << "You have got no Saving Holder" << endl;

    }

    //IF THERE IS...

    else

    {

        double amount;

        string name;

        //ENTER AMOUNT

        cout << "Enter Amount\n"

                "INPUT>> ";

        cin >> amount;

        //CHOOSES HOLDER WITH VALIDATION

        showSavingHolders();

        int num = 0;

        do

        {

            if (num != 0)

            {

                cout << "INVALID HOLDER\n";

            }

            cout << "Choose Holder\n"

                    "INPUT>> ";

            num += 1;

            cin >> name;

            transform(name.begin(), name.end(), name.begin(), ::toupper);

        } while (!savingHolderExistsValidation(name));

        //CHANGES THE VALUE TO THE NEW ONE

        NODE<Holder> \*curr = holderList.getFirst();

        while (curr != NULL)

        {

            if (name == curr->data.getName())

            {

                double finalAmount = curr->data.getAmount() + amount;

                curr->data.setAmount(finalAmount);

                //              WRITE TRANSACTION TO transactionHistory

                //              -------------------------------------------------------------------------------------------------

                createTransaction("DEP", amount, curr);

                //              -------------------------------------------------------------------------------------------------

                cout << "You have successfuly deposited money to " << name << endl;

                break;

            }

            curr = curr->link;

        }

    }

}

1. Show all holders and transaction history

void UI::showSavingHolders()

{

    double totalSaving = 0;

    mySavingHoldersHeader();

    NODE<Holder> \*curr = holderList.getFirst();

    while (curr != NULL)

    {

        stringstream ss;

        ss << curr->data.getAmount();

        string amount = ss.str();

        string name = curr->data.getName();

        string type = curr->data.getType();

        if (type == "S")

        {

            totalSaving += curr->data.getAmount();

            printf("| %-20s | %-20s |\n", name.c\_str(), amount.c\_str());

        }

        curr = curr->link;

    }

    cout << " ----------------------------------------------\n";

    cout << " TOTAL BALANCE | " << totalSaving << endl;

    cout << endl;

}

//FUNCTION TO SHOW CREDIT HOLDERS ONLY WITH TABLE

void UI::showCreditHolders()

{

    double totalCredit = 0;

    myCreditHoldersHeader();

    NODE<Holder> \*curr2 = holderList.getFirst();

    while (curr2 != NULL)

    {

        stringstream ss2;

        ss2 << curr2->data.getAmount();

        string amount2 = ss2.str();

        string name2 = curr2->data.getName();

        string type2 = curr2->data.getType();

        if (type2 == "C")

        {

            totalCredit += curr2->data.getAmount();

            printf("| %-20s | %-20s |\n", name2.c\_str(), amount2.c\_str());

        }

        curr2 = curr2->link;

    }

    cout << " ----------------------------------------------\n";

    cout << " TOTAL CREDIT | " << totalCredit << endl;

    cout << endl;

}

//FUNCTION TO PRINT TRANSACTION HISTORY WITH TABLE

void UI::showTransactionHistory()

{

    transactionHistoryHeader();

    NODE<Transaction> \*curr = transactionHistory.getFirst();

    while (curr != NULL)

    {

        string date = curr->data.getDate();

        string name = curr->data.getName();

        string type = curr->data.getTypeOfTransaction();

        stringstream ss1;

        ss1 << curr->data.getAmount();

        string amount = ss1.str();

        stringstream ss2;

        ss2 << curr->data.getBalOrCred();

        string balOrCred = ss2.str();

        stringstream ss3;

        ss3 << curr->data.getTotalBalance();

        string totalBalance = ss3.str();

        stringstream ss4;

        ss4 << curr->data.getTotalCredit();

        string totalCredit = ss4.str();

        printf("| %-10s | %-15s | %-4s | %-15s | %-15s | %-15s | %-15s |\n", date.c\_str(), name.c\_str(), type.c\_str(), amount.c\_str(), balOrCred.c\_str(), totalBalance.c\_str(), totalCredit.c\_str());

        curr = curr->link;

    }

    cout << " --------------------------------------------------------------------------------------------------------------\n";

    cout << "\*LEGEND\n"

            "PAY = PAYMENT\n"

            "DEP = DEPOSIT MONEY\n"

            "TRF = TRANSFER TO OTHER HOLDER\n"

            "CRE = DUE CREDIT PAYMENT\n"

            "NEW = INITIALIZE SAVING OR CREDIT"

         << endl;

}

**VIII. Link to GIT Website**

[https://github.com/](https://github.com/davidamadeo/Final-Project-Data-Structure-Algorithm)