ALSDD LAB1

Academic year: 2023/2024

Program Documentation

Bank Account Manager

Realized by:

- BELGUESMIA Oussama
- BELLOUL Wassim Zineedine

Groupe: 07

Problem and requirement:

The objective of this lab project is to develop an application for managing bank account processes and allow customer interactions with their accounts. The program is designed to handle various banking transactions and administrative tasks efficiently. The primary goal is to enable both customers and administrators to perform their respective functions and tasks.

Key Requirements:

1. Customer Transactions:
   - Enable customers to perform essential banking tasks, including transferring funds between accounts, depositing funds, making withdrawals, checking their account balances, and reviewing transaction history within a specified date range.
2. Administrative Functions:
   - Provide administrators with the capability to manage customer accounts effectively. This includes adding new customer accounts, deleting existing accounts, and modifying customer information as needed.
3. Store Data:
   - The application shall store all customer account information and transactions records.
   - Each customer account entry shall include account number, account code, account holder name, and account balance.
   - Transaction history shall be recorded for each account, and each banking transaction is memorized by operation code which indicates the type of transaction, date of operation, and balance credited/debited.

Design and Architecture:

Source code:

The program is composed of four header files and five C source files :

1. Header files:
   - **mainlib.h:** the main library header file contains the data structures and declarations for abstract machine functions and some other general functions.
   - **admin.h:** admin library contains declarations for functions that allow performing administrator tasks.
   - **customer.h:** customer library contains declarations for functions that allow performing customer tasks.
   - **textFileModules.h:** text file library contains declarations for functions that allow to read data from the text file and write data to the text file.
2. Source files:
   - **main.c:** the main program source file contains the main function that contains the main menu.
   - **mainlib.c:** contains the implementation for the main library functions.
   - **admin.c:** contains the implementation for functions declared in the admin library.
   - **customer.c:** contains the implementation for functions declared in the customer library.
   - **textFileModules.c:** contains the implementation for functions declared in the text file library.

Data structure:

- To store and manipulate account information we used a doubly linked list in which each node represents an account contains two address field (pointers) for the next and previous element in the linked list and the data field is a record with six fields:

1. **integer:** the account number:
2. **integer:** the account code
3. **person:** a record with two fields of type string (first name, last name) that represent the account holder.
4. **integer:** the account balance.
5. **integer**: number of account transactions.
6. **Address (pointer):** head of account transactions record linked list.

- The transaction history of each account is represented with a doubly linked list in which each node is struct with two address fields (pointers) for the next and previous element in the linked list and the data field is a record with three fields:

1. **integer:** has four possible values to indicate the operation type:
   a. 1: transfer where the account is receiver.
   b. 2: transfer where the account in sender.
   c. 3: deposit in the account.
   d. 4: withdrawal from the account.
2. **string:** date of operation (dd/mm/yy).
3. **integer**: the balance of the operation.

Abstract machine:

To manipulate the data structure we implemented an associated abstract machine with the following functions (declared in “mainlib.h” implemented in “mainlib.c”) :

- Procedures:
  - **accAllocate**: takes a pointer to an account node as parameter “p”, and allocate memory location for an account node and puts its address in “p”.
  - **assfName:** takes an account node as parameter “p”, and a string “name”, and assign “name” to first name field of “p”.
  - **asslName:** takes an account node as parameter “p”, and a string “name”, and assign “name” to last name field of “p”.
  - **accAssNext:** takes two pointers of account node “p” and “q”, and assign “q” to the “next” field of “p”.
  - **accAssPrev**: takes two pointers of account node “p” and “q”, and assign “q” to the “previous” field of “p”.
  - **tranAllocate:** takes a pointer to a transaction node “p”, and allocate memory location for a transaction node and puts its address in “p”.
  - **transAssNext:** Previous/Balance/Code: assign values to a transaction node fields
  - **tranAssCode:** assign a value to “code” field of a transaction.
  - **tranAssBalance:** assign a value to “Balance” field of a transaction.
  - **tranAssDate:** assign a value to “Date” field of a transaction.

- Functions:
  - **accNext/Prev:** returns the value of next/prev field of an account node.
  - **accCode:** returns the value of the code field of an account node.
  - **accNumber:** returns the value of the number field of an account node.
  - **accBalance:** returns the value of the Balance field of an account node.
  - **accfName:** returns a pointer to an account's first name field.
  - **accfName:** returns a pointer to an account's last name field.
  - **tranNext:** returns the value of "next" field of a transaction node.
  - **tranPrev:** returns the value of "Previous" field of a transaction node.
  - **tranCode:** returns the value of "Code" field of a transaction node.
  - **tranBalance:** returns the value of "Balance" field of a transaction node.
  - **tranDate:** returns a pointer to "date" field of a transaction node.

General Functions:

- **historyClean:** destroy transaction record linked list of an account.
- **creatNAccount:** create a linked list of N account.
- **creatNtransaction:** create a linked list of N transaction.
- **accountAccessNumber:** search for an account in the linked list by its number and return a pointer to that account.
- **accountNumberExists:** returns a boolean indicates if the account number exists in the linked list.
- **ReadUINT:** a secured input function to read an integer from the user.
- **ReadName:** read a valid name (not empty && contains only alphabet).
- **addTran:** add a transaction to an account transactions record.

Customer Functions:

- **customerSignin:** allow customer to sign in his account using account number and code.
- **customerDashboard:** a menu that allows the customer to do all banking transactions and tasks after signing in to his account.
- **customerTransfer:** allows the customer to do a transfer operation.
- **customerDeposit:** allows the customer to deposit to an account.
- **customerWithdrawal:** allows the customer to make a withdrawal of funds.
- **customerCheckBalence:** allows the customer to check his balance.
- **customerHistoryDate:** allows the customer to check his bank history in a specific date.

Administrator functions:

- **adminControlPanel:** Displays a menu that allows the admin to manage the accounts.
- **adminCreateAccount:** allows the admin to add a new account.
- **adminDeleteAccount:** allows the admin to delete an account.
- **accountEdit:** allows the admin to edit an account's informations.

Read/Write text file functions:

- **accountFromFile:** read an account and its transactions record from a text file.
- **accountToFile:** read an account and its transactions record from a text file.
- **readFile:** read all accounts from a text file and store it in a linked list.
- **writeFile:** write a linked list of accounts to a text file.

Main program (main.c):

Variables:

- AccountsN: integer indicates the number of accounts.
- Choice: used to store the user's choice from the menu.
- mainHead: the head of the accounts linked list.
- fptr: a pointer that allows to access the text file.

main function :

displays a simple menu to ask the user if he is an admin or a customer, then according to his choice, it will make a call to "adminControlPanel" or "customerDashboard" which allows him to do his tasks, or just exit the program.

Usage Instructions:

- To compile the program we attached with the source code a "makefile" which makes it possible to compile the program using the "make" command.
- After executing the "BankManager.exe" the program will automatically read accounts from the text file "acounts.txt" if found, the text file must be in the same directory as the program. then it will display a menu with three choices "1) customer", "2) admin", and "3) exit"

1. Customer: in this case the program will ask the user to enter his account number and code in order to sign in to his account, then call "customerDashboard" which allows the customer to do his tasks.
2. admin: in this case the program will ask the admin to enter the security code then if it's correct it will call "adminControlPanel" which allows the admin to do his tasks and manage the accounts.

- In order to have better control when testing the program, when exiting the program it will ask the user if he wants to save changes to the text file or not.

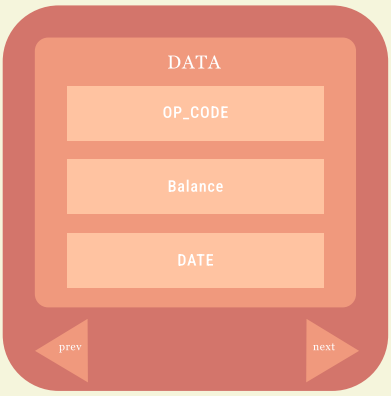
Conclusion:

- Working on the lab provided us with practical experience and taught us how to tackle real-world problems using what we have learned during classes, the best example of this is the dynamic data structure "linked list" and it's associated abstract machine we could implement the program so it can handle a large number of accounts with its transaction record allowing customers to access to their accounts and do different tasks and banking transactions, and allow the administrator to manage accounts by adding new accounts, delete accounts, and edit them.
- Building the program allowed us to learn about manipulating text files and do different operations on it read/write.
- We encountered unexpected issue during development with the user inputs which used to break our program a lot of times, but this taught us the importance of handling user inputs in a safe way.
- And also the lab provided us the opportunity to practice what we have learned about online collaboration and version control which really facilitated our work together.

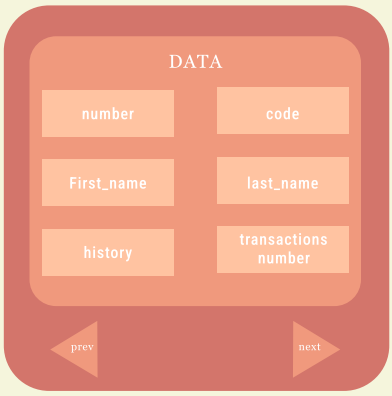
Appendices:

Our **GitHub** repository: LINK

Data Structure Diagram:



Transaction node



Accounts node

Linked list: