Design Group 3 | *Garima Maheshwari, Keziah May, Sam Wolf, Srikar Duggempudi*

Banking: Assignment 5

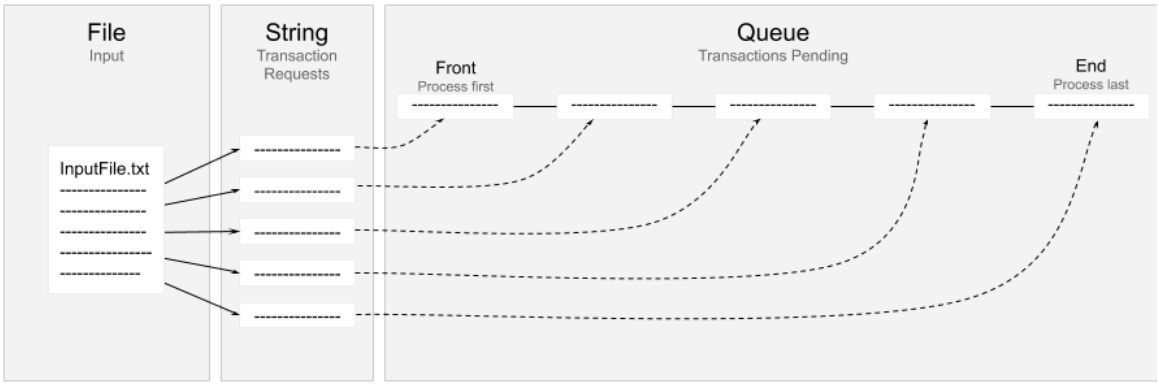
Program Design

Students of Professor Pisan's 342 class are expected to complete a bank transaction processing program as their fifth programming assignment. To assist students in the completion of the program, this document describes the design structure of a banking program which would meet the specified requirements. Provided is an overview of the program's purpose, a high-level description of the process flow, an explanation of the class interaction, a list of tests, specification for the program output, and visual models of the program, which are presented throughout the document.

Program Overview

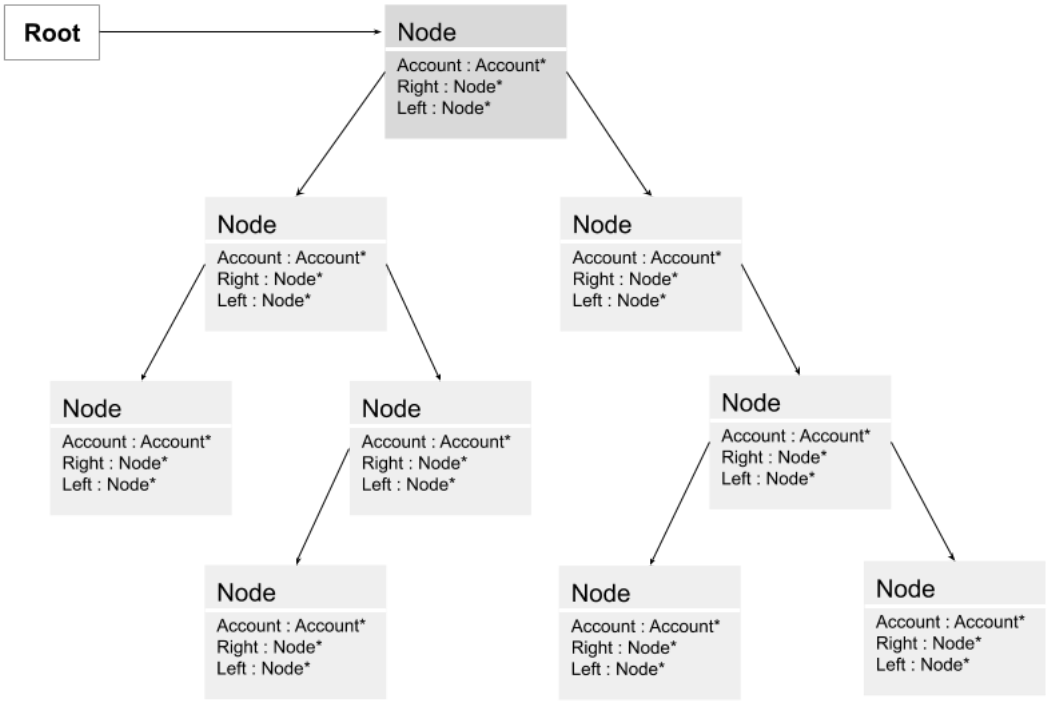
This program will take a text file containing transaction requests. Each request is stored in an STL queue as shown in **figure 1** and processed in first-in, first-out order.

Figure 1: Transaction Queue Structure Diagram



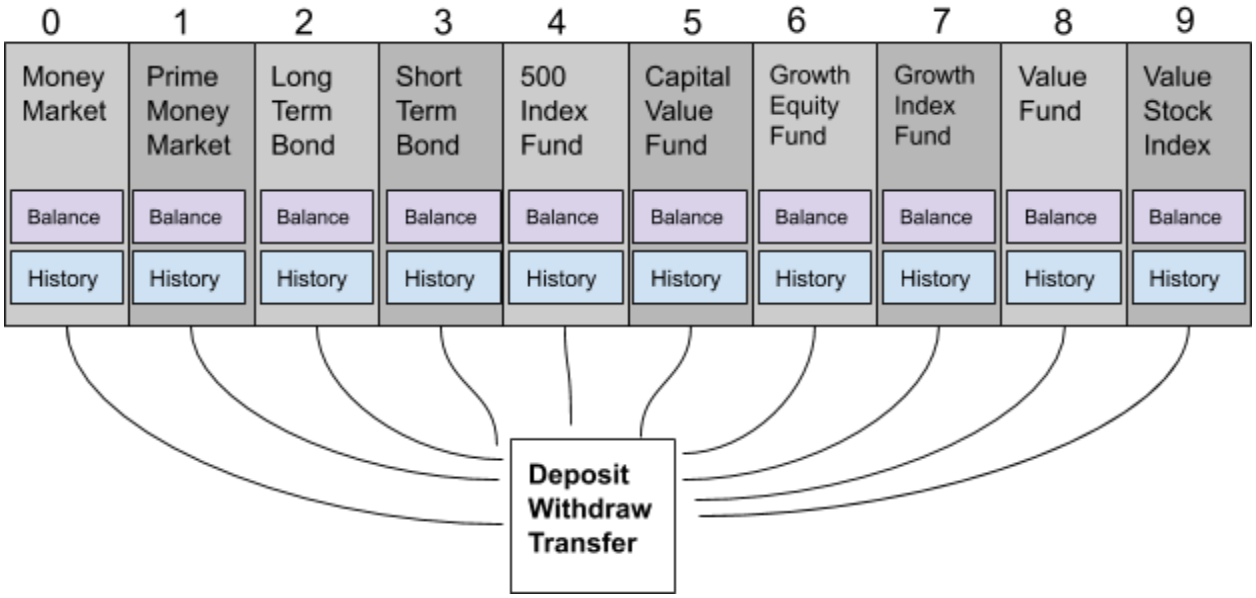
When opening an account is requested, an account is created and stored into a binary account tree (see **figure 2**).

Figure 2: AccountTree Structure Diagram



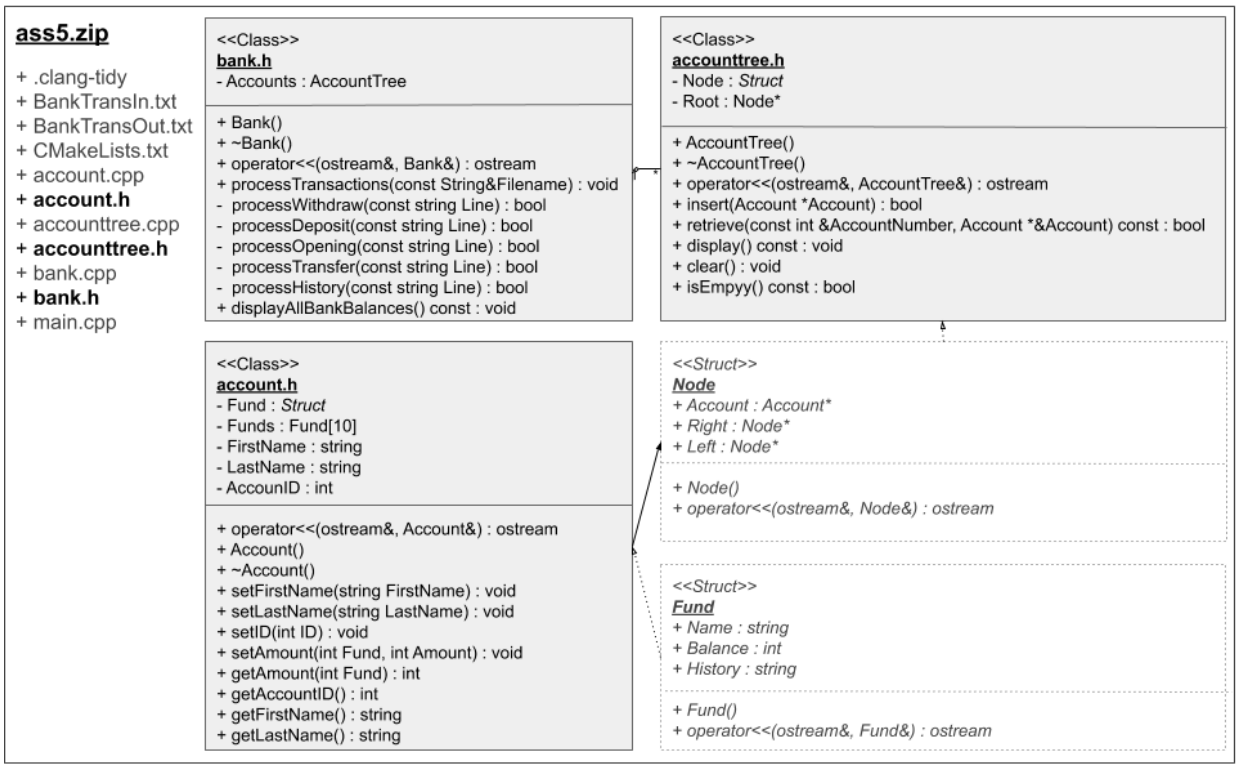
Each account will store an array of funds as seen in **figure 3.** The binary tree will have a root which is the first account that is opened in the bank. The other accounts will be ordered by their 4 digit-ID attribute either to the left (if less than root ID) or to the right (if greater than root value).

Figure 3: Structure Diagram of the Array of Funds found in account.h



Once an account has been opened, deposits, withdrawals, transfers, and transaction history requests can be attempted for the account using the bank class defined in **figure 4**.

Figure 4: Package Diagram of ass5.zip



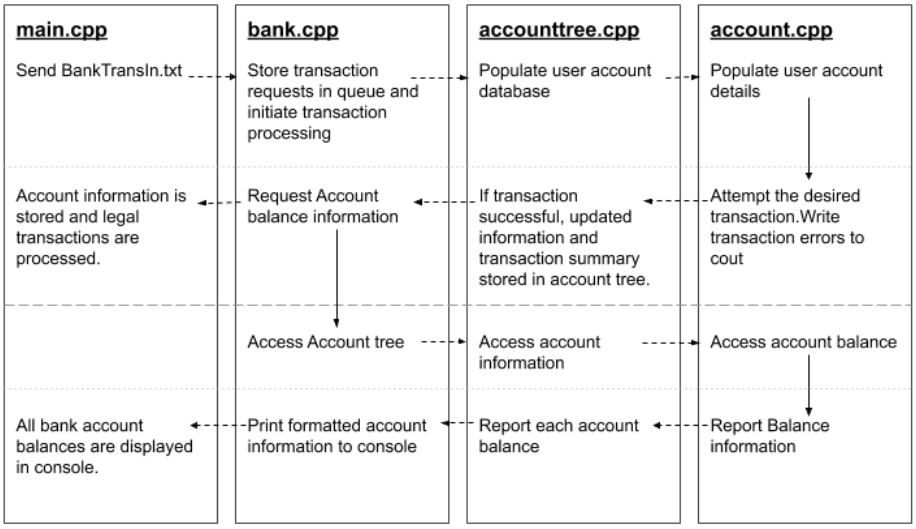
To process a deposit, use the “processDeposit” function. The first four digits of the given account ID are used to search the tree for the account. The rightmost digit specifies which account fund receives the deposit by index.
To withdraw money, use the “processWithdraw” function. After the account is located, when there are sufficient funds in the account, the amount is withdrawn. When there are not sufficient funds an error message will be sent to cout and the account state will be unchanged. This is true with one exception, when withdrawing from a money market or bond account, if there are insufficient funds, the difference can be taken from a fund of the same type (bond to bond).
To transfer money, use the “processTransfer” function. The origin account is treated like a withdrawal. If successful, the destination account is treated like deposit.
To request a transaction summary, use the “processHistory” function. The fund history stored in the given account(s) will be reported to cout.

The final phase of our program is to print the account balances for all accounts in the tree. To validate the program, the test cases described in the “test cases” section below should be run. Memory leaks should be checked by running the valgrind command on Linux, to ensure dynamically allocated tree has been deallocated completely.

Process Flow

This section provides a trace of the flow of information when an input file with the expected formatting is sent to main.cpp.

Figure 5: Process Diagram of ass5.zip



High-level Process Requirements

An input file of transaction requests is sent to the program. As seen in **figure 5**, the file is passed to a bank which organizes the transactions in the order they were requested and initiates the processing of each transaction in first-in, first-out order. Each account created is added to an account database and transaction requests on those accounts are processed if possible. If there is an error processing the request, the state of the account is unchanged, and the error is reported to the console. If the transaction was successful, the account information is updated and the transaction summary is logged. Once each transaction has been processed, the account database is accessed to report the balances of each account. Each account and associated balance are printed to the console.

Class Interaction

The program consists of 3 classes: Bank, Account and AccountTree. The transaction input file is sent from main to the Bank class. The Bank processes the transactions one line at a time. Each transaction goes to the AccountTree class to either look for the Account to apply the transaction or add a new Account. The Account class makes appropriate transaction changes. Once these changes are made, the AccountTree gets updated with those changes. The Bank class continues to process the rest of the transactions until the end of the file is reached. At the end, the Bank class goes through the AccountTree and prints all the Accounts with their balances to the console.

Testing

Input

This program will take a text file with transaction requests formatted as follows:

Deposits: `"D <Account ID : int> <Dollar amount : int>"`
Withdrawls: `"W <Account ID : int> <Dollar amount : int>"`
Transfers: `"T <Account ID : int> <Dollar amount : int> <Account ID : int>"`
Transaction History: `"H <Account ID : int>"`
Open new Account: `"O <First Name: string> <Last Name: string> <Account ID : int>"`

- Account ID:
    - If four digits, refers to an account which contains up to 10 fund accounts.
    - If five digits, the rightmost digit refers to a specific fund associated with the account ID defined by the first four digits.
- The input items in "<>" braces should be replaced with the given parameter and the braces should be removed.
- Each request ends with a newline.

Process Testing

The following tests are to be used to validate the banking program does what it is supposed to.

Main.cpp tests using Assert(Bool) or Assert(StringStream == Expected Output<String>)

```
// Test Queue to make sure it does in-order transaction reading
testTransactionReading()

// Test displayOutput matches "Output Design"
testDisplayOutput()

// Test insert account into the BinaryTree and that behavior is correct
testInsertAccount()

// Test retrieving account from Binary Tree and that behavior is correct
testRetrieveAccount()

// Test that a list of strings will be correctly handled and processed
testProcessTransactions()

// Assert correct behavior on account
testProcessWithdrawl()

// Assert correct behavior on account
testProcessDeposit()

// Assert correct behavior on account
testProcessOpening()

// Assert correct behavior on account
testProcessTransfer()

// Assert correct behavior on account
testProcessHistory()
```

```
// Test that an isEmpty reports correct bool results
testIsEmpty()
```

Output Design

cout << Bank

```
Returns Out << TransactionTree;
```

cout << AccountTree;

```
recursive loop {

Out << Account

}
```

cout << Account

```
Account <FirstName> <LastName> <ID#>

<FundName>: <FundBalance>

<FundName>: <FundBalance>

...
```

Display a State-Mutating Transaction

```
<TransactionType> <ID#> <FundName>: <BalanceDifferenceInDollars>
```