30/10/2022

48430 | Fundamentals of C Programming

Assignment 3: Group Project

Lab: 01
Group: 03

Mohammed Khan - 13940236
Zhicheng Lin - 13505375
Chloe Kernaleguen - 13851581

Fundamentals of C Programming - Assignment 03

Introduction

For this assignment, we have decided to focus on the financial technologies sector by developing a command-line-based banking system written in the C programming language. The goal of this system is to create a safe & secure way to handle individual bank accounts, transactions & balances in a lightweight manner whilst ensuring data confidentiality by encrypting critical information such as login pins and passwords with the RSA algorithm. We additionally hope to compress & decompress extremely long transactions/user files in order to allow our banking application to save on storage as well as further obfuscate important information.

I. Objective, Scope and Program Features

Objective

In this assignment, we will develop a banking system in the C programming language with the ability to compress, decompress, encrypt & decrypt data. To achieve this, we will be building a command-line-based application that has the ability to interact with user account information stored in text files. User information will be displayed in a formatted manner, and users will be able to select multiple menu options to manipulate this information as well as create additional users.

As we wish for this application to become secure & lightweight, we will be using the Huffman compression algorithm to compress the users' account data before storing it in the text file. In addition, we will be using the RSA encryption algorithm to encrypt & decrypt key user information, including their password & associated pin code.

Scope

As this assignment is primarily assessed on encryption & compression, the scope of this application will be primarily focused on implementing these two features. The user account information will be read from a text file containing account details such as account number, account username, account balance, transaction history etc. For account information files with extremely large sizes, the application will be able to compress these files using the Huffman algorithm. However as compression is not entirely secure, and any malicious user could easily decompress the file to obtain user information. The application will also encrypt critical information such as the users' password & pin using the RSA algorithm to ensure the file is secure from any malicious attacks.

The application will also provide a command-line interface for users to interact with their account information. Users will be able to create an account, log in to their account and then subsequently view their account balance, transaction history, account details etc. In addition, users will also be able to make deposits & withdrawals, all with associated input validation & security through the use of confirmation PINs prior to withdrawals. All this information will be appended/added to the associated users' text file to ensure that account operations are persistent.

Validation checks, additionally, will be put in place for user inputs, with the application prompting the user to re-enter their input should it be deemed invalid.

Program Features

The banking application will have the following features:

1. User account creation: Users will be able to create an account by providing their preferred username, password & associated 4-character long payment code (PIN)
2. User login: Users will be able to login into their account by providing their corresponding account number & associated password. This password will be decrypted by the program and stored in memory to prevent the user from reading the stored password in plain text
3. View account information: Once logged in, users will be able to view their account information, including their: accountID, username, password, payment code (PIN), balance, number of transactions & transaction history
4. Deposit money: Logged-in users will be able to make deposits of positive decimal values into their account by providing the amount of money they wish to deposit
5. Withdraw money: Logged-in users will be able to make withdrawals from their accounts by providing a positive decimal value of the amount they wish to withdraw as well as their corresponding 4-digit confirmation PIN. After this PIN is inputted correctly, money will be withdrawn from the account.
6. Encryption: The application will encrypt sensitive user account information using the RSA algorithm prior to storing it in the associated users' text file
7. Decryption: The application will decrypt the users' information upon the loading of this application using the RSA algorithm. The decrypted information will be stored in memory and will not be visible to the user in any associated text files for security reasons
8. Compression: Large account files with extremely large amounts of transactions can be additionally compressed for better storage using the Huffman compression algorithm
9. Decompression: Large account files that are already compressed can be decompressed when needed to be used using the Huffman decompression algorithm
10. Logout: Logged-in users will be able to sign out of their account & return back to the main menu, where they can log into another account or create a new account

Through these features, the application will provide a functional & secure banking system for the user to store & manage their account information.

II. Project Architecture / Design

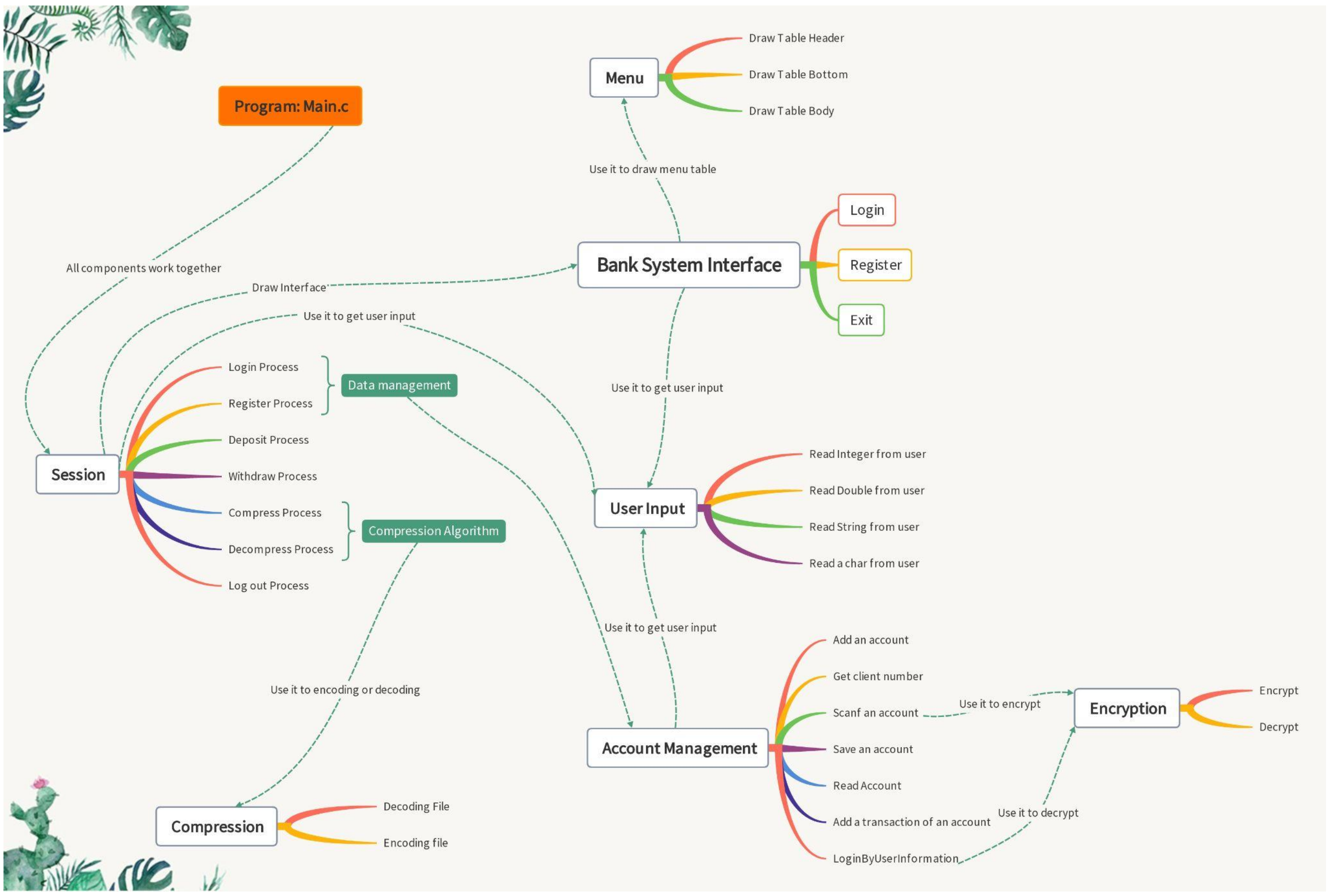
Project Design

The goal of this application is to build a feature-rich application that is loosely coupled in order to make it easier to maintain, extend and develop with multiple developers. In order to achieve this, we have designed the architecture of the project in such a way that each component can work independently of the other and can be developed in parallel removing development blockers in accordance with the INVEST criteria for development. (Bigger Impact, 2022)

In order to achieve this, the application will be designed with a modular approach. To achieve this, the application will be divided into separate modules. Each will be responsible for a separate task. For example, *encryption.c* will be the module responsible for handling the RSA encryption/decryption methods, whilst another would be handling compression/decompression using the Huffman algorithm.

By designing the software in this way, and making each module its self-contained unit independent of other modules. This makes our application easier to swap out one module for another if necessary i.e changing RSA to a more robust algorithm if a better one is found or improving our compression algorithm. This in addition, also simplifies our testing methodology as each module can be tested independently of each other, minimizing the risk of side effects affecting our code and allowing these modules to be fully complete before being integrated into the larger project.

A diagram of our proposed architecture explaining how each module will interact with each other is shown below:

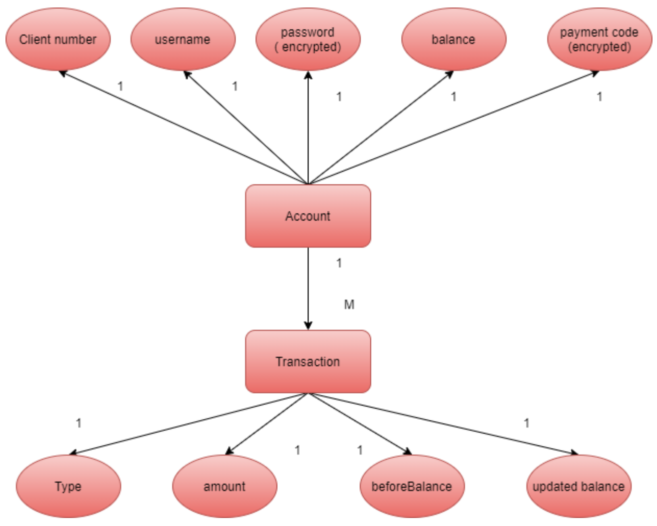


Menu

The menu is the primary component responsible for rendering the user interface in a table-like format, with input prompts also provided and formatted in the same way. The purpose of the menu component is to render the table header, table body and footer in a consistent and DRY manner, preventing the excessive repetition of *printf* statements to format inputs. The menu, in addition to rendering the table, also provides formatted color to input prompts, allowing for a consistent color scheme. *Figure 1: Main Menu*

Account Management

The account management component provides a series of functions/methods for working with our persistent data store, which is in the form of a text file. This method allows us to create, read, update and delete user accounts as well as provides us the ability to encrypt & decrypt the users' passwords & PINs with our RSA algorithm when saving or storing to our text file. Through this account management file, our application automatically appends and saves to the text file every time an operation is successfully performed, i.e., once a deposit is confirmed, the deposit is automatically added to the associated account file. *Figure 2: Account Management Struct*



Entity Relationship Diagram of the User Structure

Session

The session component contains the majority of our business logic. It is the module responsible for managing multiple key operations in our application, including:

- Login Process
- Register a new account
- Logout
- Deposit money
- Withdraw money
- Compress user information
- Decompress user information
- View account/transaction details

These functions additionally call a series of sub-functions included in *userInput.c* that contains the logic for validating & retrieving user input, preventing issues from buffer overflow, spaces, incorrect data types etc. *Figure 3: Login, Figure 4: Register, Figure 5: Logout, Figure 6: Deposit, Figure 7: Withdraw, Figure 8: Compress User Information, Figure 9: Decompress User Information, Figure 10: View User Information*

Encryption

Unlike the Huffman compression algorithm described below. The encryption algorithm was significantly easier to implement. In terms of design, the encryption algorithm is split into two distinct parts, key generation & encryption.

Key Generation

The key generation process is where the public & private keys are generated, which are then used to encrypt and decrypt messages, respectively. The encryption method takes in a message and character by character converts the ASCII value to an integer and uses the public key to encrypt the message into a ciphertext. Conversely, the decryption algorithm takes in the private key and uses the private key values to decode the cipher text into the original message by reversing the encryption process.

To implement this algorithm, I first started by selecting two large prime numbers *p & q*. Due to me wanting to keep the encryption method consistent & able to decode previous encoded messages, I decided that the values of *p & q* would be constant for the entire program (p = 37, q = 31). With these values set, I then calculated the values of *n,* which is *p * q*, and *phi(n),* which is *(p - 1)*(q - 1). Figure 11: Initialize Encryption*

Next, I generated the values for e, which is one of the values of the public key, such that n was relatively prime to *phi(n).* This meant they would have a *gcd(e, d(n)) = 1.* Where e was also *1 < e < phi(n).* With *e* calculated, I now had the public key, which is a tuple consisting of <*n, e*>. *Figure 12: Generate Public Key <e, n>*

For the private key, the value of d was generated such that *(d * e) % phi(n) = 1.* This meant that the value of d had to be *1 < d < phi(n).* To calculate this, I iterated through a for loop, until a value that satisfied this condition was calculated. After determining the value of *d,* I then also had the private key, which was a tuple consisting of *<n, d>. Figure 13: Generate Private Key <d, n>*

Encryption

After determining the public key. The encryption process was able to be easily implemented. This was done by taking in a message, converting it into a character array and then casting each character to its corresponding ASCII value. These values were then encrypted by the RSA algorithm with the following equation:

C = (m ^ e) % n, where c is the cipher text, m is the ASCII value of the message, e is the public key value, and n is the product of p & q. *Figure 14: Encrypt Message (Returns int[])*

Decryption

The decryption process was similarly implemented after determining the private key. This was done by taking in the encrypted message (the ciphertext), reading the ciphered message in its integer array form and then reversing the encryption process using the following equation:

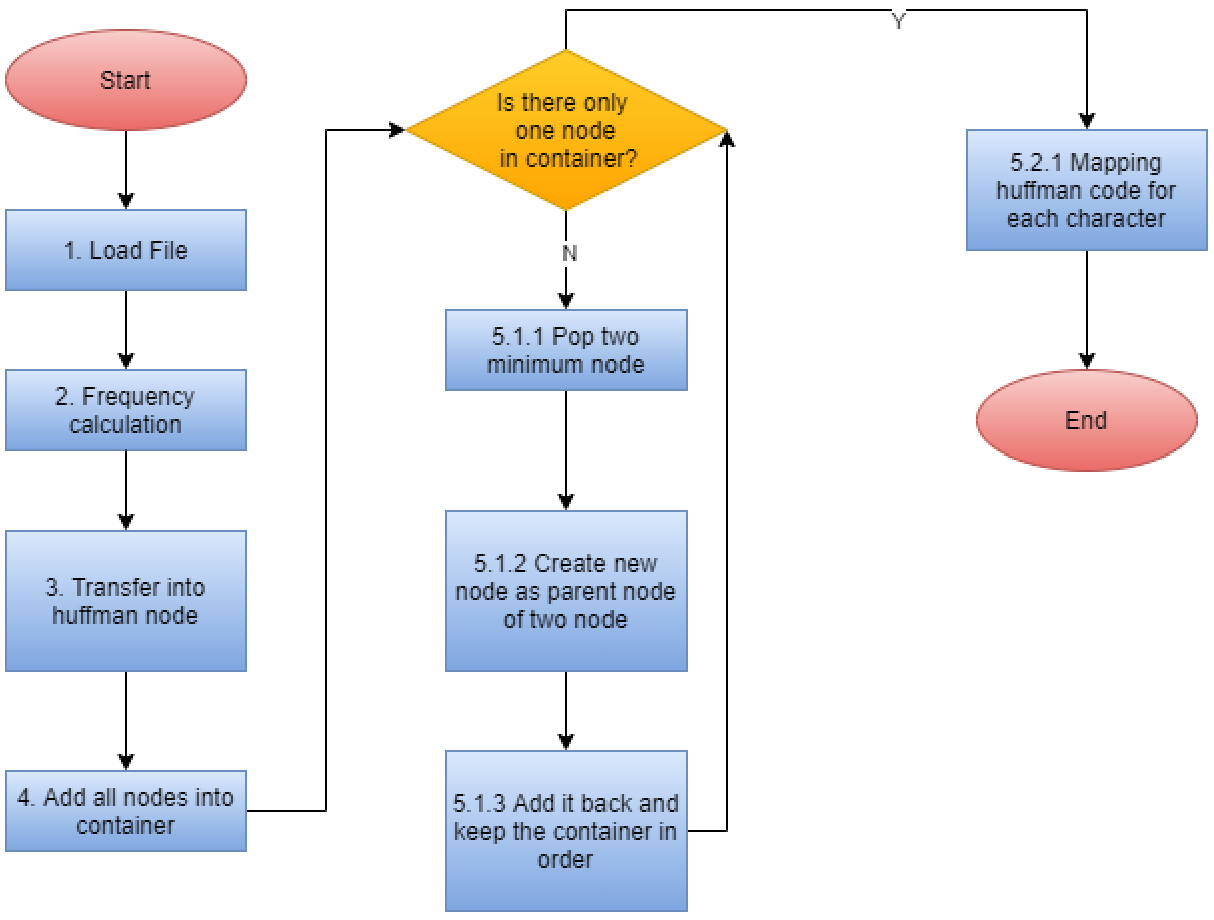
M = (c ^ d) % n, where m is the original message, c is the ciphertext, d is the private key value, and n is the product of p & q. *Figure 15: Decrypt Message (Returns char[])*

By implementing these processes above in C, I was able to encrypt and decrypt a message using the RSA algorithm successfully. (Javatpoint, 2022)

Compression

In compression, after exploring several other compression algorithms, such as run-length encoding and determining them to not fit our use-case, Huffman compression was ultimately decided as the algorithm of choice to implement for the compression section.

Huffman encoding is a variable-length compression method that works by forming a frequency table based on the frequency of each occurrence of a specific character. Huffman encoding aims to achieve compression by assigning as few binary codes as possible to the most frequently occurring characters. The central idea of Huffman compression is to create a custom Huffman tree for a file and to use the tree to generate the encoded data. The flowchart below shows the steps involved in building & using the Huffman tree. (Stanford, 2022)


Start
1. Load File
2. Frequency calculation
3. Transfer into huffman node
4. Add all nodes into container
Is there only one node in container?
Y
N
5.1.1 Pop two minimum node
5.1.2 Create new node as parent node of two node
5.1.3 Add it back and keep the container in order
5.2.1 Mapping huffman code for each character
End

Flowchart of how Huffman Algorithm Works

As shown in the diagram above, a detailed explanation of each step is given below:

1. The application first loads the source file into memory.
2. The application then scans the source data to calculate the frequency of each character
3. Each character is transferred into a Huffman node based on the results given in the frequency table. The Huffman node is a binary tree that contains two child elements and internally contains the character and the frequency count.

Helloworld

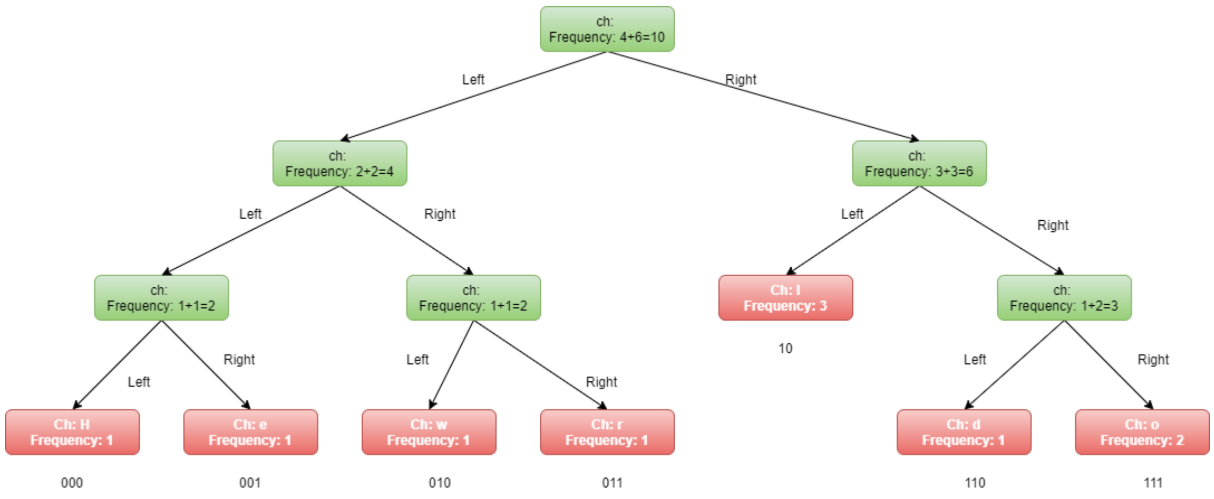
Character	**H**	**e**	**w**	**r**	**d**	**o**	**l**
Frequency	1	1	1	1	1	2	3

Table depicting how Huffman uses frequency analysis of characters in its algorithm

4. All Huffman nodes are then added to a container, and the container orders the nodes in accordance with the frequency. To implement this, we chose a minimum heap as the container to keep order.

5. The system now uses the heap to build the Huffman tree
   5.1 The steps shown below repeat recursively until the size of the minimum heap is of length 1.
      5.1.1 The system will first pop two nodes, with the lowest frequency found in the heap
      5.1.2 The system then creates a new node as the parent node, and the frequency of this new node, will be the sum of the two lowest frequency nodes, popped in step 5.1.1. The left child node & right child node of this node will also be the two nodes selected in step 5.1.1
      5.1.3 The system then adds this new node back to the heap. The heap will rearrange all the nodes.
   5.2 Once the process is complete, where the size of the minimum heap has reached 1. The process will then get the root of the Huffman tree. As the tree is now built, all the characters will now be encoded based on the position of the character in the Huffman tree. By navigating through the tree, with left nodes being labeled as 0 and right nodes being labeled as 1. As we traverse through the tree we will replace the associated character with its new binary form. I.e. the character H will be replaced with 000 as it is the leftmost node. (Example listed, in the diagram below)

By using this process, we're able to significantly reduce the file size and compress the file.



Frequency is then organized into a binary tree, using a graph data structure

III. Problems Encountered & Justification of Design Choices

Encryption

Finding a way to securely encrypt user information

For this assignment, we initially started by researching multiple encryption methods, initially starting with trying to implement ROT-13 or a substitution/caesar cipher.

Why initial attempts were ineffective

After conducting further research, we determined that ROT-13 was ineffective as it was easily reversible and had a pattern that could ease the deciphering process by looking at the ciphertext. We also found that a substitution cipher/caesar cipher was also as equally ineffective as it was incredibly easy to find the offset the text was encrypted with.

What we did instead

After discovering that our initial methods were insecure, we decided to find a more secure way of encrypting user information, and as a result, we decided to use the RSA encryption method. RSA was effective as due to its asymmetric nature, it was very secure, and due to its reliance on prime numbers and modulus operations, which by themselves are also difficult to factor, it has a statistically impossible chance to be cracked without the private key, with the time taken being longer than the expected lifespan of the universe.

Allowing Unlimited Transactions Per User

The number of transactions a user makes is uncertain

An account can have a varying number of transactions, and as a result, it isn't possible to predict how many transactions a user will make with this program. As a result of this, we've decided to store our user transactions in an array to make this almost unlimited. We've set the transaction limit of this array to 5000.

Problems encountered with making transactions lists in an array

By hardcoding the number of transactions per account, it is possible to make the array too small for a specific user and cause the program to crash due to an overflow error. It is also possible to make the array too large and therefore waste memory space, causing the program to consume too much memory.

Solution to the problem

To solve this problem, we decided to replace our array with a linked list to allocate memory space and resize the array when needed dynamically. By creating a linked list, and resizing it with the malloc, the system was able to significantly save memory space by only reserving memory when required, in addition to eliminating the risk of buffer overflow errors by allocating more space when required. (David Frame, 2021)

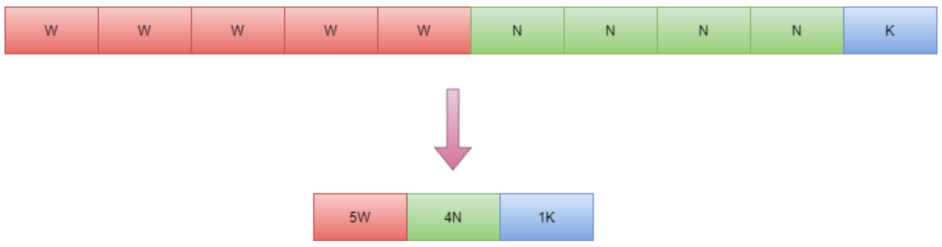
Compression

Finding a way to compress information

We initially attempted compression using the run-length encoding technique to compress the account statements. However, we quickly realized that the run-length encoding technique was ineffective for our solution as the compressed file was always larger than the initial source file.

Why the run-length encoding technique was ineffective

Run-length encoding works by compressing a consecutive sequence of repeating symbols and replacing it with the number of times it occurs. I.e., AAAAA = 5A. Whilst this is effective in some situations, as the account information stored was typically short in length and primarily a sequence of numbers that did not repeat. As a result, the run-length encoding technique failed to compress our account statements and instead added excess data, being larger than the original file. (Wikipedia, 2022)



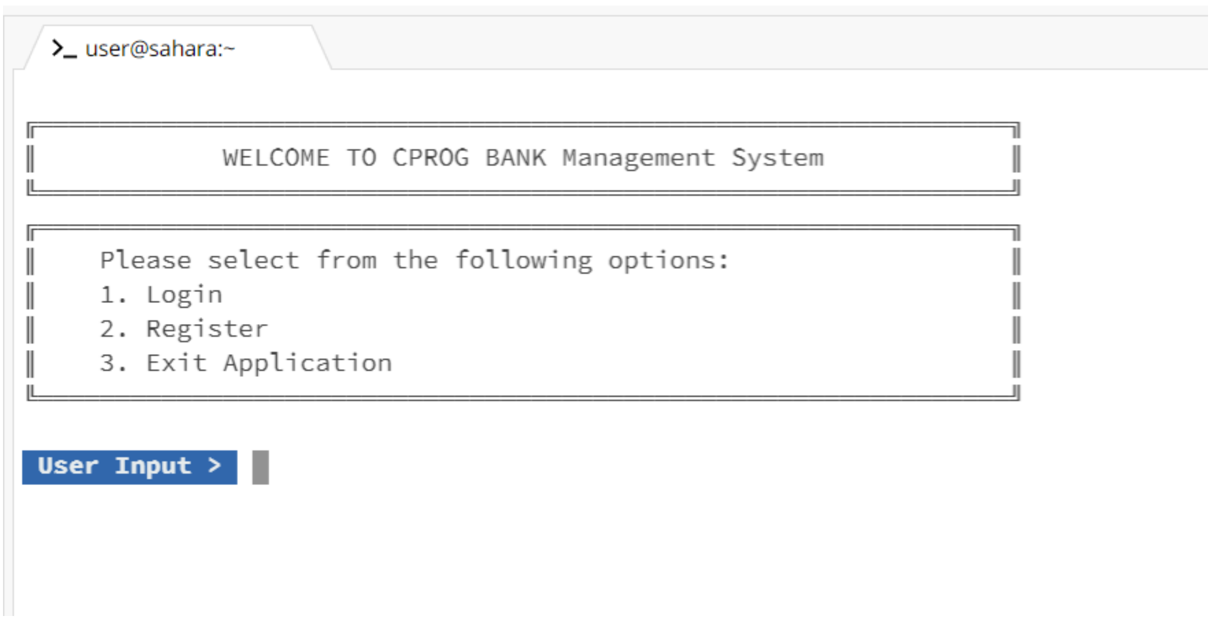
Solution to this problem

As described in the design section, we ultimately ended up implementing the Huffman encoding method. Huffman, instead of counting the number of consecutive symbols instead, works by determining the frequency of a character. After determining the frequency, it then assigns a code to the symbol. As a result of this, we are able to compress text more effectively without the same issues run-length encoding had. (Stanford, 2022)

Bibliography

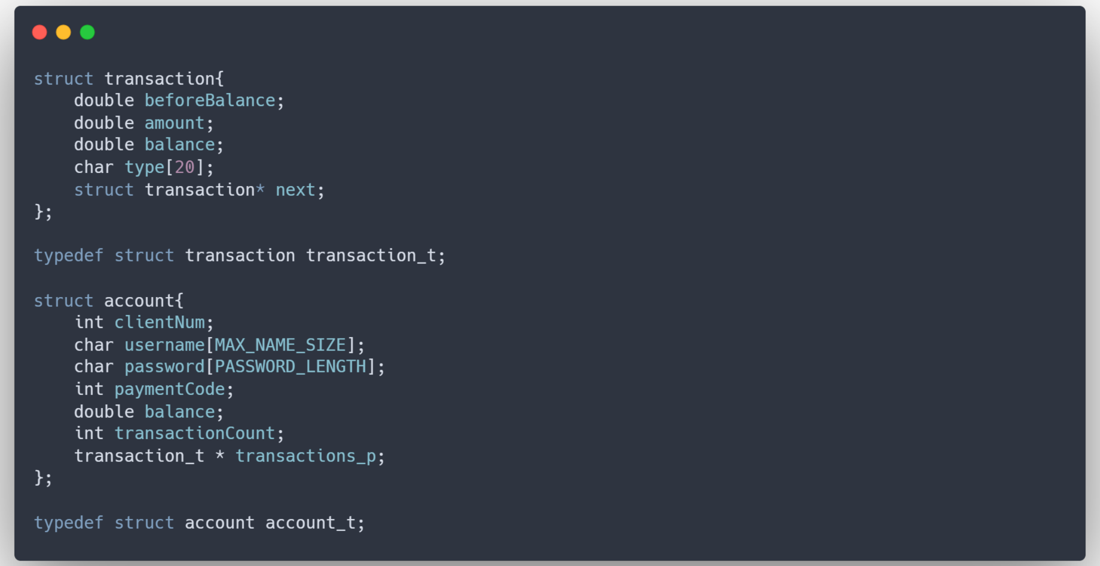
- *INVEST criteria for Agile user stories | Bigger Impact* (2022). Available at: https://www.boost.co.nz/blog/2021/10/invest-criteria (Accessed: 5 November 2022).
- *RSA Encryption Algorithm - Javatpoint* (2022). Available at: https://www.javatpoint.com/rsa-encryption-algorithm (Accessed: 5 November 2022).
- *Run-length encoding - Wikipedia* (2022). Available at: https://en.wikipedia.org/wiki/Run-length_encoding (Accessed: 5 November 2022).
- *Huffman Algorithm* (2022). Available at: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/huffman/algorithm.htm (Accessed: 5 November 2022).
- *On Complexity: Linked Lists vs. Arrays* (2021). Available at: https://devdaveframe.medium.com/on-complexity-linked-lists-vs-arrays-290b364514c3 (Accessed: 5 November 2022).

Appendix



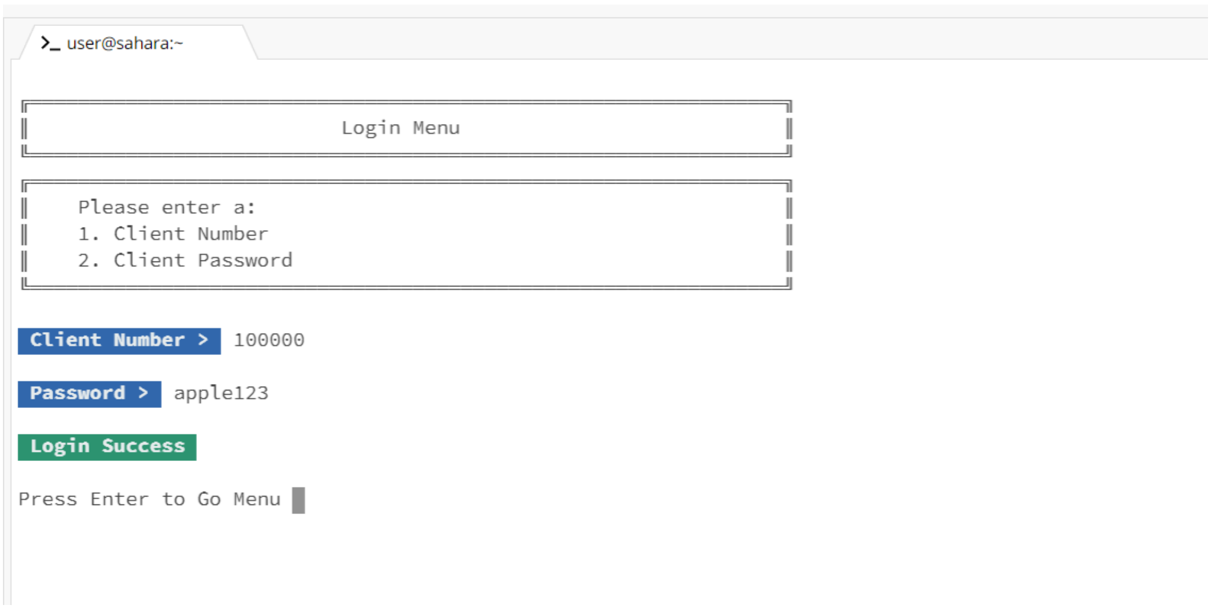
```
>_ user@sahara:~

╔══════════════════════════════════════════════════════════════════╗
║             WELCOME TO CPROG BANK Management System              ║
╚══════════════════════════════════════════════════════════════════╝

╔══════════════════════════════════════════════════════════════════╗
║     Please select from the following options:                    ║
║     1. Login                                                     ║
║     2. Register                                                  ║
║     3. Exit Application                                          ║
╚══════════════════════════════════════════════════════════════════╝

User Input >
```

Figure 1: Main Menu



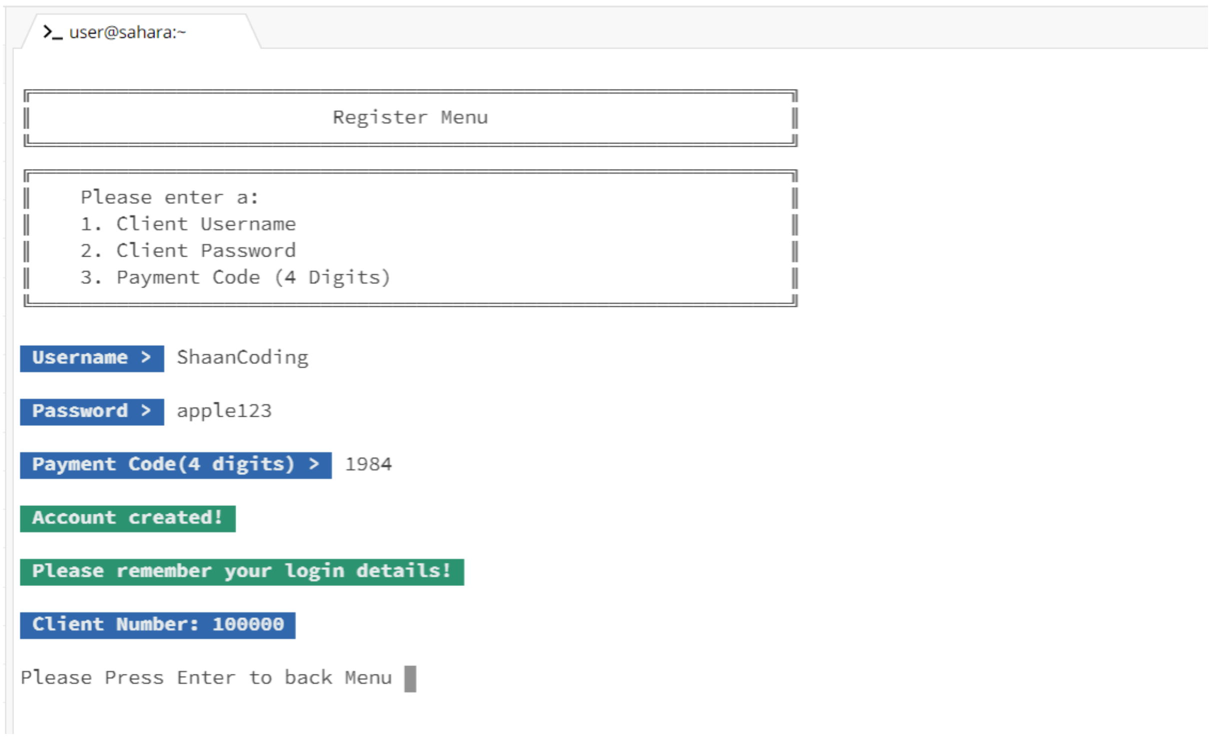
```
struct transaction{
    double beforeBalance;
    double amount;
    double balance;
    char type[20];
    struct transaction* next;
};

typedef struct transaction transaction_t;

struct account{
    int clientNum;
    char username[MAX_NAME_SIZE];
    char password[PASSWORD_LENGTH];
    int paymentCode;
    double balance;
    int transactionCount;
    transaction_t * transactions_p;
};

typedef struct account account_t;
```

Figure 2: Account Management Struct



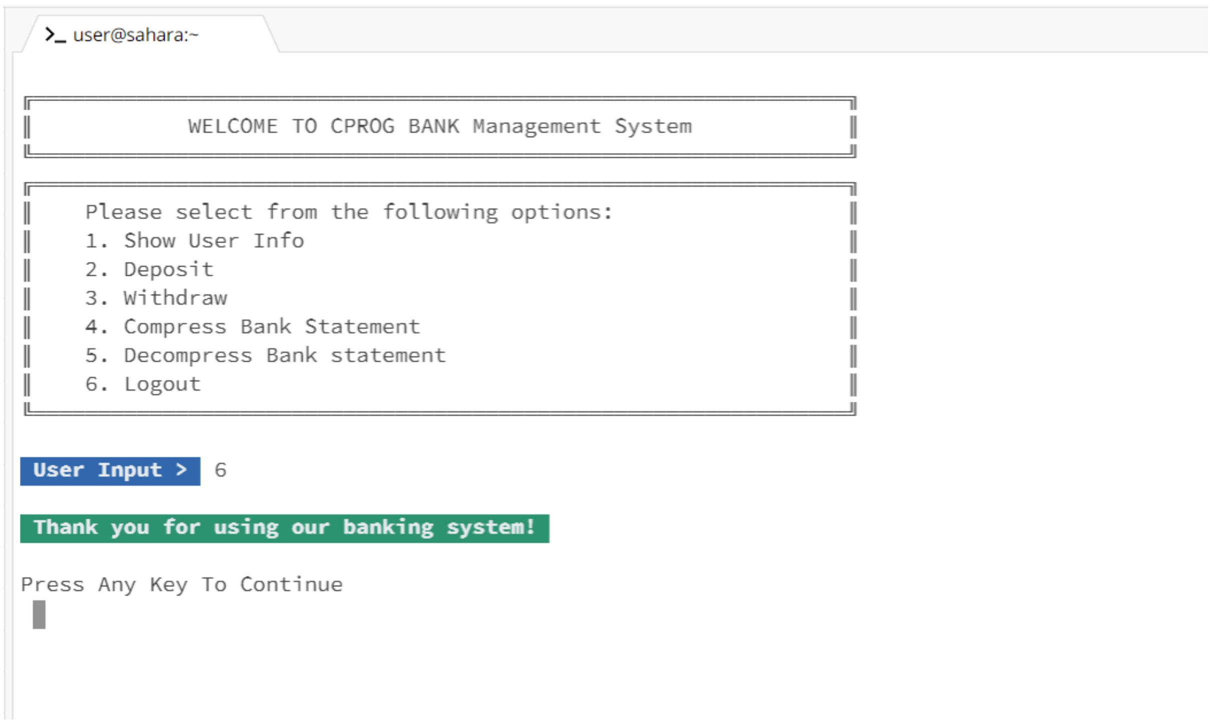
```
>_ user@sahara:~

╔══════════════════════════════════════════════════╗
║                    Login Menu                    ║
╚══════════════════════════════════════════════════╝
╔══════════════════════════════════════════════════╗
║    Please enter a:                               ║
║    1. Client Number                              ║
║    2. Client Password                            ║
╚══════════════════════════════════════════════════╝

Client Number >  100000

Password >  apple123

Login Success

Press Enter to Go Menu
```

Figure 3: Login



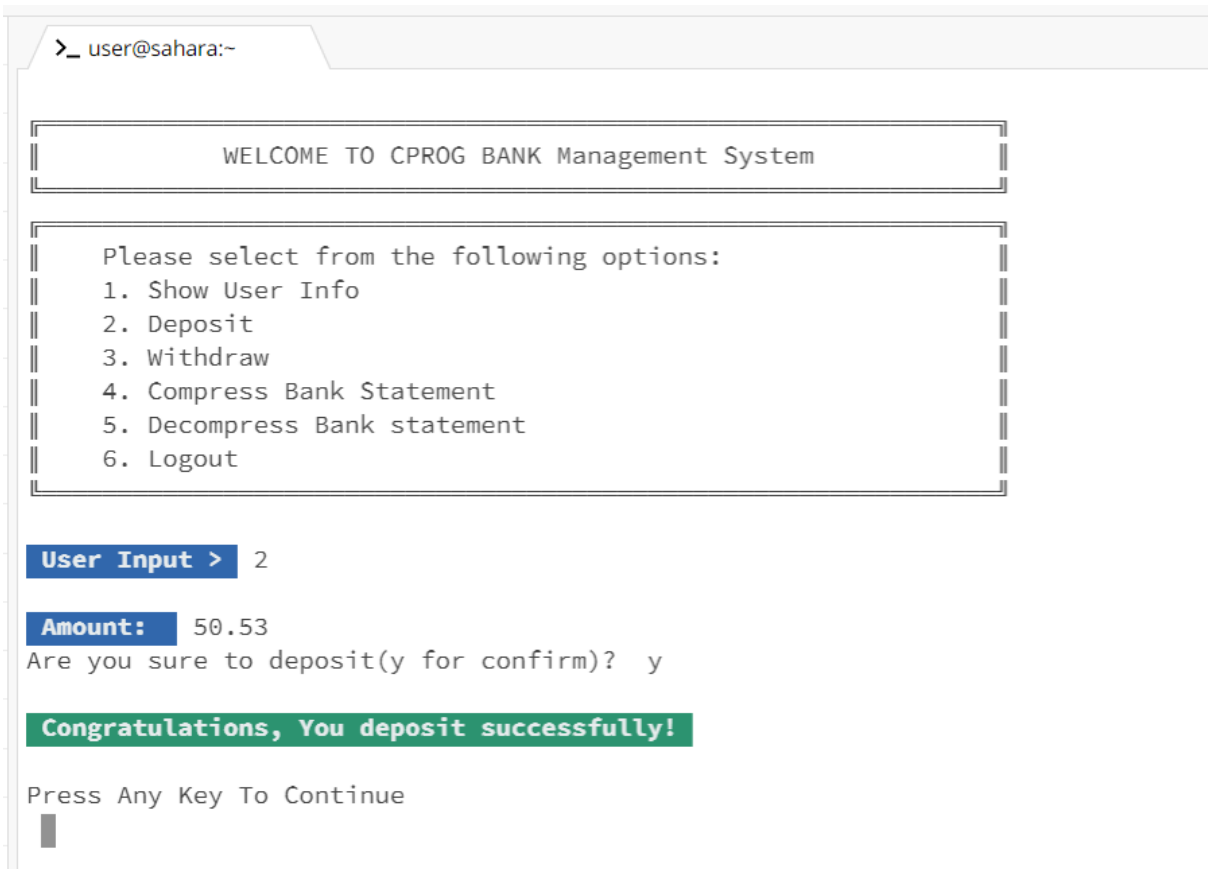
```
>_ user@sahara:~

╔══════════════════════════════════════════════════╗
║                   Register Menu                  ║
╚══════════════════════════════════════════════════╝
╔══════════════════════════════════════════════════╗
║    Please enter a:                               ║
║    1. Client Username                            ║
║    2. Client Password                            ║
║    3. Payment Code (4 Digits)                    ║
╚══════════════════════════════════════════════════╝

Username >  ShaanCoding

Password >  apple123

Payment Code(4 digits) >  1984

Account created!

Please remember your login details!

Client Number: 100000

Please Press Enter to back Menu
```

Figure 4: Register



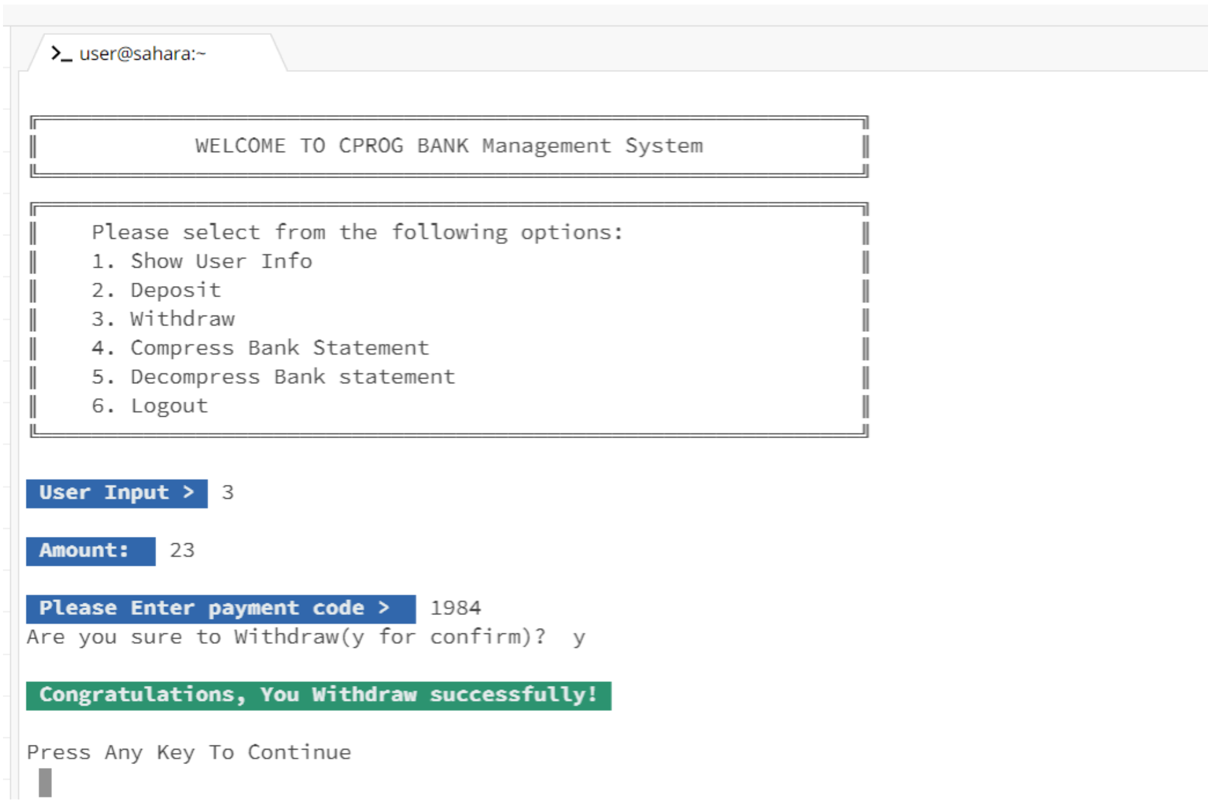
```
>_ user@sahara:~

╔══════════════════════════════════════════════════════════════╗
║              WELCOME TO CPROG BANK Management System         ║
╚══════════════════════════════════════════════════════════════╝

╔══════════════════════════════════════════════════════════════╗
║    Please select from the following options:                 ║
║    1. Show User Info                                         ║
║    2. Deposit                                                ║
║    3. Withdraw                                               ║
║    4. Compress Bank Statement                                ║
║    5. Decompress Bank statement                              ║
║    6. Logout                                                 ║
╚══════════════════════════════════════════════════════════════╝

 User Input >  6

 Thank you for using our banking system!

Press Any Key To Continue
```

Figure 5: Logout



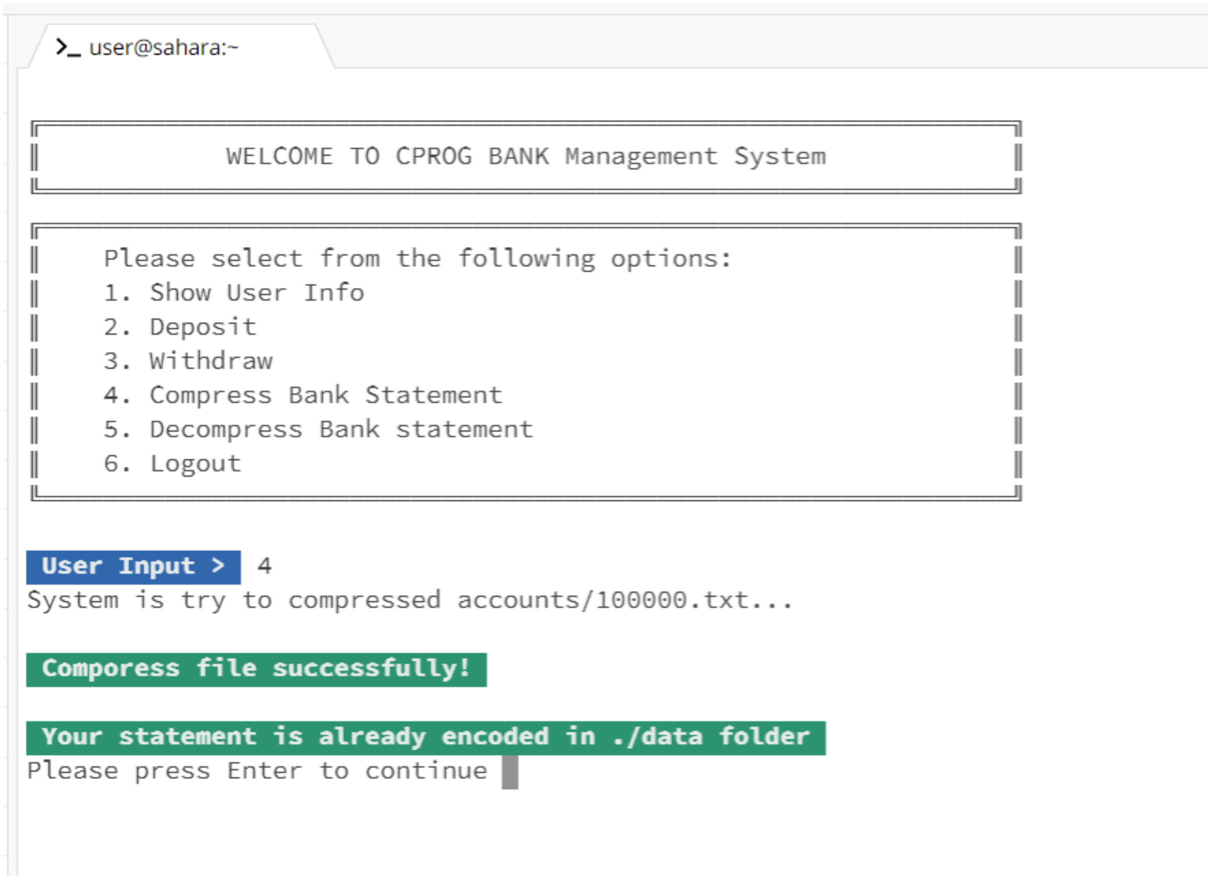
```
>_ user@sahara:~

╔══════════════════════════════════════════════════════════════╗
║              WELCOME TO CPROG BANK Management System         ║
╚══════════════════════════════════════════════════════════════╝

╔══════════════════════════════════════════════════════════════╗
║    Please select from the following options:                 ║
║    1. Show User Info                                         ║
║    2. Deposit                                                ║
║    3. Withdraw                                               ║
║    4. Compress Bank Statement                                ║
║    5. Decompress Bank statement                              ║
║    6. Logout                                                 ║
╚══════════════════════════════════════════════════════════════╝

 User Input >  2

 Amount:   50.53
Are you sure to deposit(y for confirm)?  y

 Congratulations, You deposit successfully!

Press Any Key To Continue
```

Figure 6: Deposit



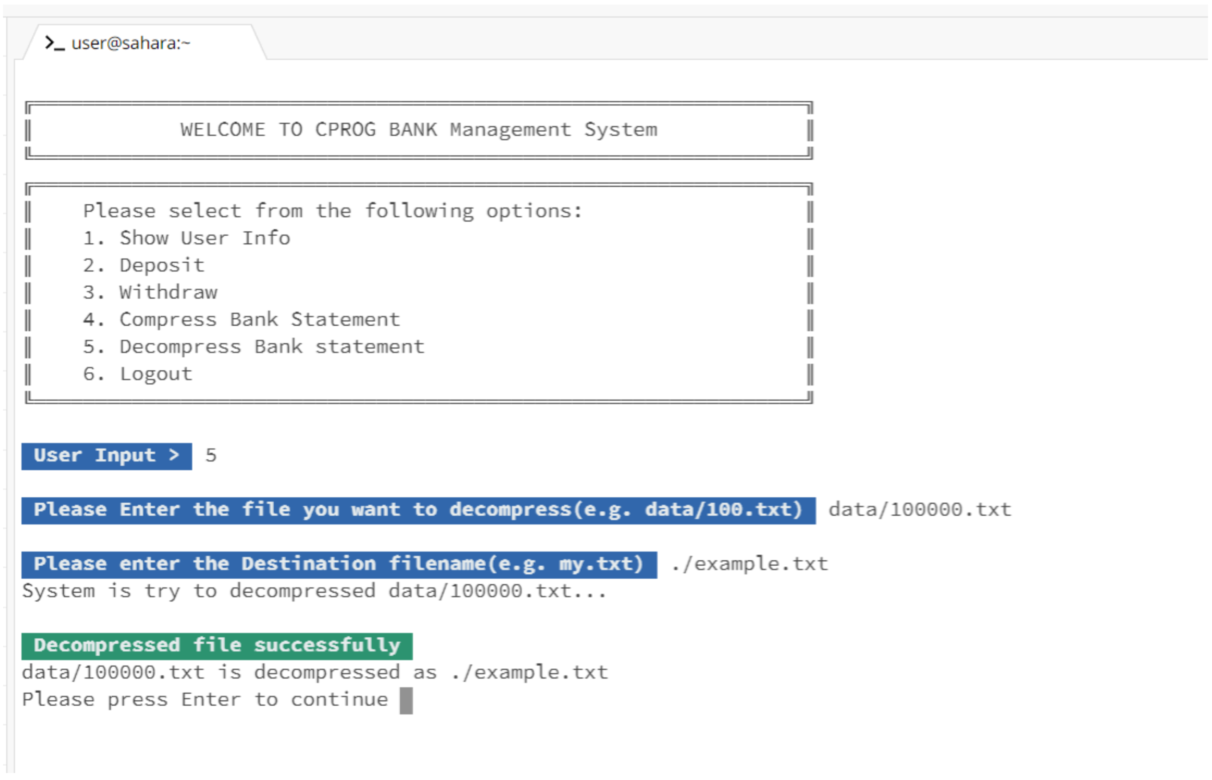
```
>_ user@sahara:~

╔══════════════════════════════════════════════════════════════╗
║               WELCOME TO CPROG BANK Management System          ║
╚══════════════════════════════════════════════════════════════╝
╔══════════════════════════════════════════════════════════════╗
║    Please select from the following options:                   ║
║    1. Show User Info                                           ║
║    2. Deposit                                                  ║
║    3. Withdraw                                                 ║
║    4. Compress Bank Statement                                  ║
║    5. Decompress Bank statement                                ║
║    6. Logout                                                   ║
╚══════════════════════════════════════════════════════════════╝

User Input >  3

Amount:  23

Please Enter payment code >  1984
Are you sure to Withdraw(y for confirm)?  y

Congratulations, You Withdraw successfully!

Press Any Key To Continue
```

Figure 7: Withdraw



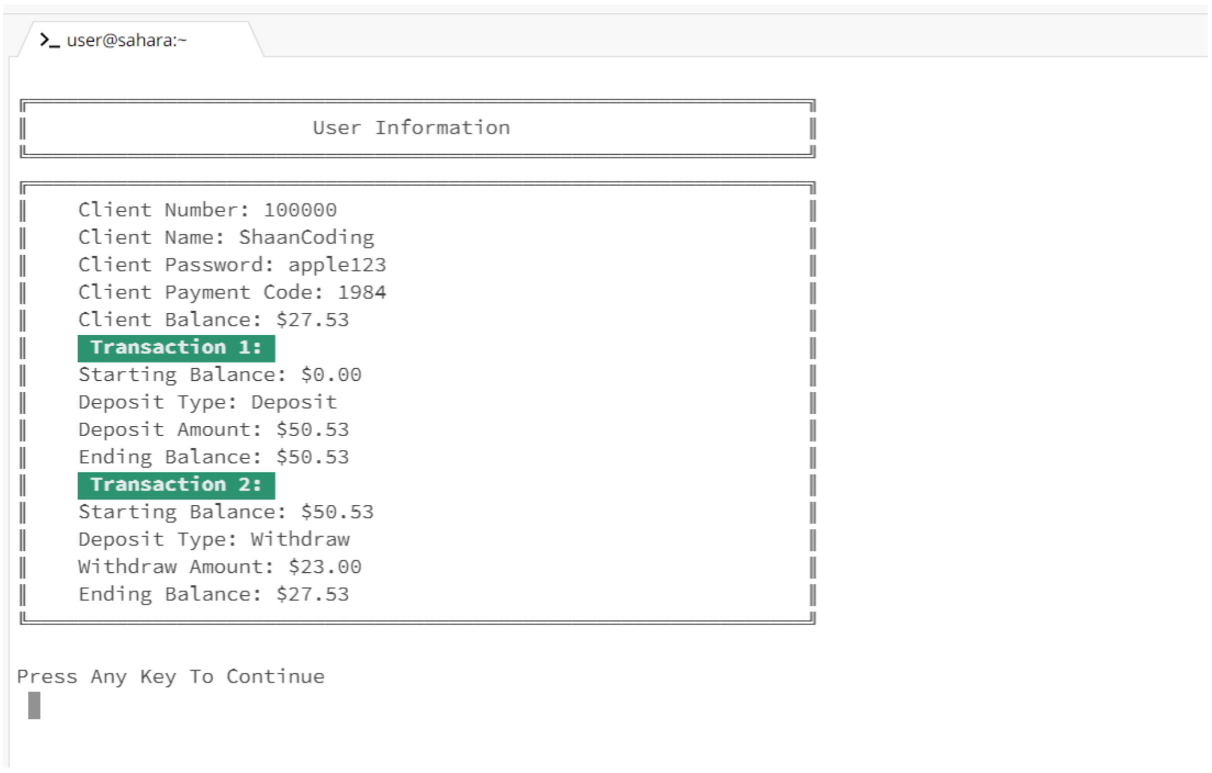
```
>_ user@sahara:~

╔══════════════════════════════════════════════════════════════╗
║               WELCOME TO CPROG BANK Management System          ║
╚══════════════════════════════════════════════════════════════╝
╔══════════════════════════════════════════════════════════════╗
║    Please select from the following options:                   ║
║    1. Show User Info                                           ║
║    2. Deposit                                                  ║
║    3. Withdraw                                                 ║
║    4. Compress Bank Statement                                  ║
║    5. Decompress Bank statement                                ║
║    6. Logout                                                   ║
╚══════════════════════════════════════════════════════════════╝

User Input >  4
System is try to compressed accounts/100000.txt...

Comporess file successfully!

Your statement is already encoded in ./data folder
Please press Enter to continue
```

Figure 8: Compress User Information



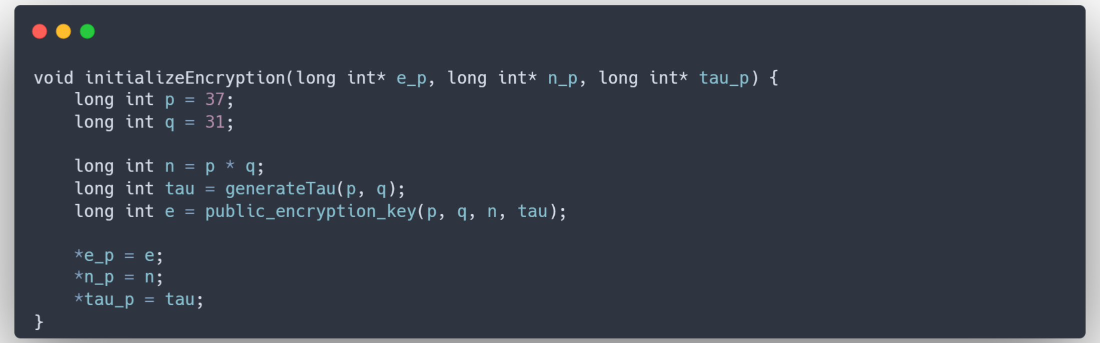
```
>_ user@sahara:~

                WELCOME TO CPROG BANK Management System

        Please select from the following options:
        1. Show User Info
        2. Deposit
        3. Withdraw
        4. Compress Bank Statement
        5. Decompress Bank statement
        6. Logout

User Input > 5

Please Enter the file you want to decompress(e.g. data/100.txt) data/100000.txt

Please enter the Destination filename(e.g. my.txt) ./example.txt
System is try to decompressed data/100000.txt...

Decompressed file successfully
data/100000.txt is decompressed as ./example.txt
Please press Enter to continue
```

Figure 9: Decompress User Information



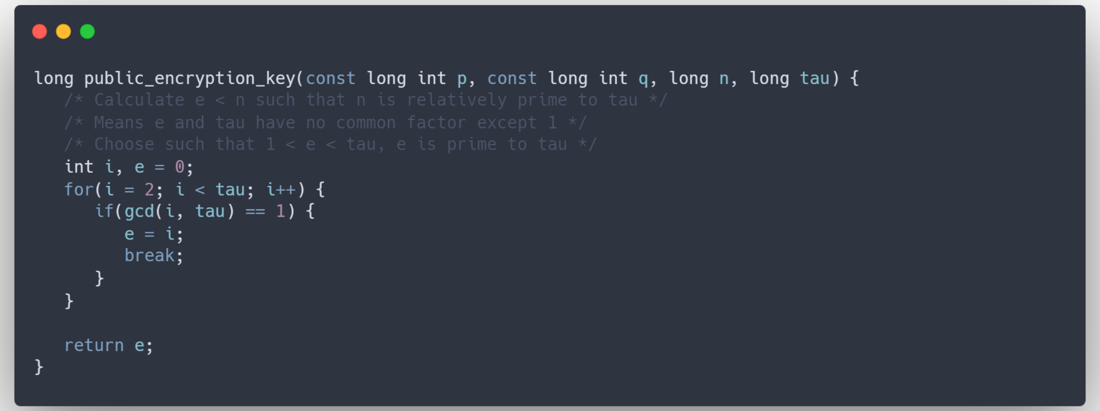
```
>_ user@sahara:~

                         User Information

        Client Number: 100000
        Client Name: ShaanCoding
        Client Password: apple123
        Client Payment Code: 1984
        Client Balance: $27.53
         Transaction 1:
        Starting Balance: $0.00
        Deposit Type: Deposit
        Deposit Amount: $50.53
        Ending Balance: $50.53
         Transaction 2:
        Starting Balance: $50.53
        Deposit Type: Withdraw
        Withdraw Amount: $23.00
        Ending Balance: $27.53

Press Any Key To Continue
```

Figure 10: View User Information



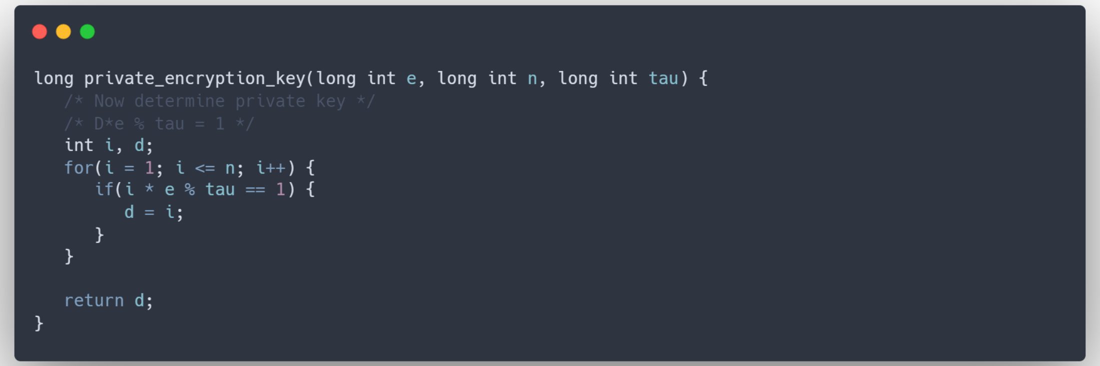
```c
void initializeEncryption(long int* e_p, long int* n_p, long int* tau_p) {
    long int p = 37;
    long int q = 31;

    long int n = p * q;
    long int tau = generateTau(p, q);
    long int e = public_encryption_key(p, q, n, tau);

    *e_p = e;
    *n_p = n;
    *tau_p = tau;
}
```

Figure 11: Initialize Encryption



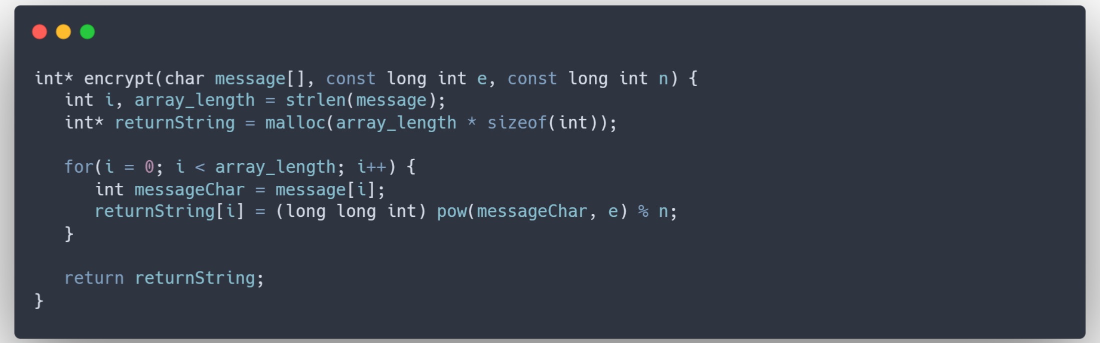
```c
long public_encryption_key(const long int p, const long int q, long n, long tau) {
   /* Calculate e < n such that n is relatively prime to tau */
   /* Means e and tau have no common factor except 1 */
   /* Choose such that 1 < e < tau, e is prime to tau */
   int i, e = 0;
   for(i = 2; i < tau; i++) {
      if(gcd(i, tau) == 1) {
         e = i;
         break;
      }
   }

   return e;
}
```

Figure 12: Generate Public Key <e, n>



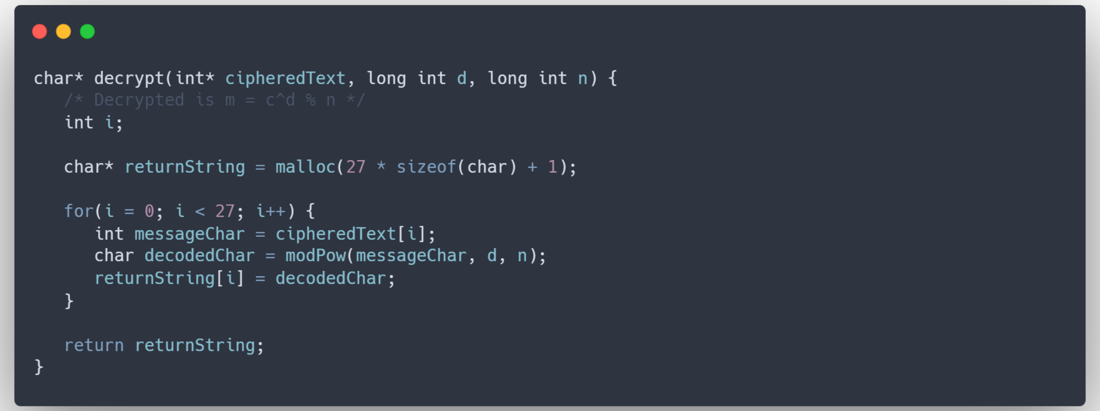
```c
long private_encryption_key(long int e, long int n, long int tau) {
   /* Now determine private key */
   /* D*e % tau = 1 */
   int i, d;
   for(i = 1; i <= n; i++) {
      if(i * e % tau == 1) {
         d = i;
      }
   }

   return d;
}
```

Figure 13: Generate Private Key <d, n>



```c
int* encrypt(char message[], const long int e, const long int n) {
   int i, array_length = strlen(message);
   int* returnString = malloc(array_length * sizeof(int));

   for(i = 0; i < array_length; i++) {
      int messageChar = message[i];
      returnString[i] = (long long int) pow(messageChar, e) % n;
   }

   return returnString;
}
```

Figure 14: Encrypt Message (Returns int[])



```c
char* decrypt(int* cipheredText, long int d, long int n) {
   /* Decrypted is m = c^d % n */
   int i;

   char* returnString = malloc(27 * sizeof(char) + 1);

   for(i = 0; i < 27; i++) {
      int messageChar = cipheredText[i];
      char decodedChar = modPow(messageChar, d, n);
      returnString[i] = decodedChar;
   }

   return returnString;
}
```

Figure 15: Decrypt Message (Returns char[])

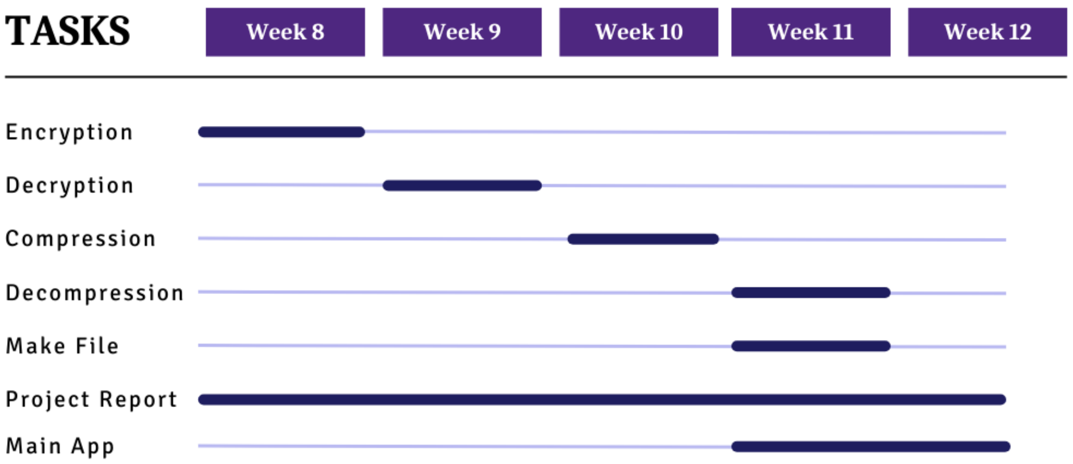
Schedule (Appendix Content)

The timeline for this project is detailed in the Gantt chart below. The program has been divided into five main functions: Encryption, Decryption, Compression & Decompression, with additional tasks for the main page and application interactions listed as separate, minor tasks. Creating a Makefile as well as writing the report are also listed & scheduled in the Gantt below.

In the first two weeks, from weeks 8 to 9, the focus will be on the research & implementation of the compression & decompression algorithm in addition to generating the report. In weeks 10 to 11, the focus will be on implementing encryption and decryption as well as finalizing compression. By the start of week 11 and the end of week 12, the final tasks of linking the application together into the main program will also be completed. The remainder of the time will be spent on finalizing the report & focusing on minor grammatical errors.

The report, in addition to weeks 8 & 9, will be generated throughout the project as a whole and is a continuous part of the entire sprint from weeks 8 to 12.

FUNDAMENTALS OF C PROGRAMMING PROJECT TIMELINE


TASKS
Week 8
Week 9
Week 10
Week 11
Week 12
Encryption
Decryption
Compression
Decompression
Make File
Project Report
Main App