Financial Tracker with a Circular Doubly LinkedList in C

Aung Htin Kyaw

March 9, 2025

Contents

1 Introduction

This report provides an overview of the Income and Expense Tracker program. The program is designed to help users track their income and expenses, providing a clear view of their financial status. The program is implemented in C and uses a **doubly linked list** to manage transactions. The report will explain the key components of the program, including the linked list implementation, the user interface, and the main program logic.

2 Doubly Linked List Implementation (linkedlist.c)

The core of the program is the **doubly linked list** implementation in `linkedlist.c`. A doubly linked list is a data structure that consists of nodes, where each node contains data and pointers to both the previous and next nodes in the sequence. This allows for efficient traversal in both directions, making it easier to insert, delete, and manage nodes.

Node Structure Each node in the doubly linked list contains the following information:

- `type`: The type of transaction (income or expense).
- `description`: A description of the transaction.
- `amount`: The amount of money involved in the transaction.
- `status`: The status of the transaction (e.g., new, saved, marked for deletion).
- `prev`: A pointer to the previous node in the list.
- `next`: A pointer to the next node in the list.

The list is circular, meaning the last node points back to the first node, and the first node points to the last node. This is achieved using a **sentinel node**, which acts as a dummy node to simplify list operations.

—

Key Functions in linkedlist.c

1. **createNode()** This function creates a new node and initializes its fields. It allocates memory for the node and sets its `prev` and `next` pointers. The function is used whenever a new transaction is added to the list.

```
Node* createNode(Node* prev, const char* type, const char*
    description, double amount, char* status, Node* next) {
  Node *new_node = (Node*)malloc(sizeof(Node));
  if (new_node == NULL) {
    printf("Memory allocation failed\n");
    exit(1);
  }

  new_node->type = strdup(type); // Copy the type string
```

```
  new_node->description = strdup(description); // Copy the
    description string
  new_node->amount = amount; // Set the transaction amount
  new_node->status = strdup(status); // Copy the status string

  new_node->prev = prev; // Set the previous node pointer
  new_node->next = next; // Set the next node pointer

  new_node->size = 0; // Initialize size (used for sentinel node)
  new_node->balance = 0; // Initialize balance (used for sentinel
    node)
  return new_node;
}
```

—

2. **addFirst()** This function adds a new node to the **beginning** of the linked list. If the list is empty, it creates the first node and sets it as both the head and tail. Otherwise, it updates the pointers of the existing nodes to accommodate the new node.

```
void addFirst(Node* sen, const char* type, const char* description,
     double amount) {
  char status[30] = "(new)"; // Default status for new transactions
  if (sen->size == 0) { // If the list is empty
    Node* new_node = createNode(sen, type, description, amount,
    status, sen);
    sen->next = new_node; // Sentinel points to the new node
    sen->prev = new_node; // Sentinel points to the new node (
    circular list)
  }
  else { // If the list is not empty
    Node* firstItem = sen->next; // Get the current first node
    Node* new_node = createNode(sen, type, description, amount,
    status, firstItem);
    firstItem->prev = new_node; // Update the previous pointer of
    the first node
    sen->next = new_node; // Update the sentinel's next pointer
  }
  sen->balance += amount; // Update the balance
  sen->size += 1; // Increment the size of the list
}
```

—

3. **addLast()** This function adds a new node to the **end** of the linked list. If the list is empty, it calls `addFirst()` to add the node. Otherwise, it updates the pointers of the last node and the sentinel node to accommodate the new node.

```
void addLast(Node* sen, const char* type, const char* description,
    double amount) {
  char status[30] = "(new)"; // Default status for new transactions
  if (sen->size == 0) { // If the list is empty
    addFirst(sen, type, description, amount); // Use addFirst to
    add the node
  }
  else { // If the list is not empty
    Node* lastItem = sen->prev; // Get the current last node
```

```
    Node* new_node = createNode(lastItem, type, description, amount
    , status, sen);
    lastItem->next = new_node; // Update the next pointer of the
    last node
    sen->prev = new_node; // Update the sentinel's previous pointer
    sen->balance += amount; // Update the balance
    sen->size += 1; // Increment the size of the list
  }
}
```

—

4. **insert()** This function inserts a new node at a **specific position** in the linked list. It traverses the list to find the correct position and updates the pointers of the surrounding nodes to accommodate the new node.

```
void insert(Node* sen, const char* type, const char* description,
    double amount, int pos) {
  int position = pos - 1; // Convert position to zero-based index
  if (sen->size == 0 || position == 0) { // If the list is empty or
     inserting at the beginning
    addFirst(sen, type, description, amount);
  }
  else if (position == sen->size) { // If inserting at the end
    addLast(sen, type, description, amount);
  }
  else if (position < 0 || position > sen->size) { // Invalid
    position
    printf("Invalid position\n");
    return;
  }
  else { // Inserting in the middle of the list
    Node* current = sen;
    for (int i = 0; i < position; i++) { // Traverse to the desired
     position
      current = current->next;
    }
    char status[30] = "+++ i"; // Status for inserted nodes
    Node* nextNode = current->next; // Get the next node
    Node* new_node = createNode(current, type, description, amount,
     status, nextNode);
    current->next = new_node; // Update the current node's next
    pointer
    nextNode->prev = new_node; // Update the next node's previous
    pointer
    sen->balance += amount; // Update the balance
    sen->size += 1; // Increment the size of the list
  }
}
```

—

5. **markDelete()** This function marks a node for deletion by changing its status to '— d'. It does not physically remove the node from the list but updates its status and adjusts the balance.

```
void markDelete(Node* sen, int pos) {
  if (sen->size == 0) { // If the list is empty
    return;
```

```
  }
  Node* current = sen->next; // Start from the first node
  for (int i = 0; current != sen && i < pos - 1; i++) { // Traverse
     to the desired position
    current = current->next;
  }
  if (current == sen) { // If the position is out of range
    return;
  }
  free(current->status); // Free the old status
  current->status = strdup("—— d"); // Mark the node for deletion
  sen->balance -= current->amount; // Adjust the balance
}
```

—

6. **update()** This function physically removes all nodes marked for deletion from the list. It traverses the list, removes the marked nodes, and updates the pointers of the surrounding nodes.

```
void update(Node* sen) {
  if (sen->size == 0) { // If the list is empty
    return;
  }
  Node* current = sen->next; // Start from the first node
  while (current != sen) { // Traverse the list
    if (strcmp(current->status, "—— d") == 0) { // If the node is
    marked for deletion
      Node* tmp = current; // Store the node to be deleted
      current = current->next; // Move to the next node
      if (tmp->prev != NULL) { // Update the previous node's next
    pointer
        tmp->prev->next = tmp->next;
      }
      if (tmp->next != NULL) { // Update the next node's previous
    pointer
        tmp->next->prev = tmp->prev;
      }
      free(tmp); // Free the node
      sen->size -= 1; // Decrement the size of the list
      continue;
    }
    current = current->next; // Move to the next node
  }
}
```

—

7. **printList()** This function prints all transactions in the list, along with their descriptions, amounts, and statuses. It is used to display the current state of the list to the user.

```
void printList(Node* sen) {
  printf("  ******************  \n");
  printf("  **  Transactions **  \n");
  printf("  ******************  \n");
  int labelNum = 1;

  if (sen->size != 0) { // If the list is not empty
```

```
    Node* current = sen->next; // Start from the first node
    do {
      printf("%d. %s\t%.2f\t%s\n", labelNum, current->description,
    current->amount, current->status);
      current = current->next; // Move to the next node
      labelNum++;
    } while (current != sen); // Continue until the sentinel is
    reached
    printf("\n");
  }
  else { // If the list is empty
    printf("Empty List\n");
  }
}
```

—

Visualization of Doubly Linked List Operations Below is a visualization of how the doubly linked list operations work:

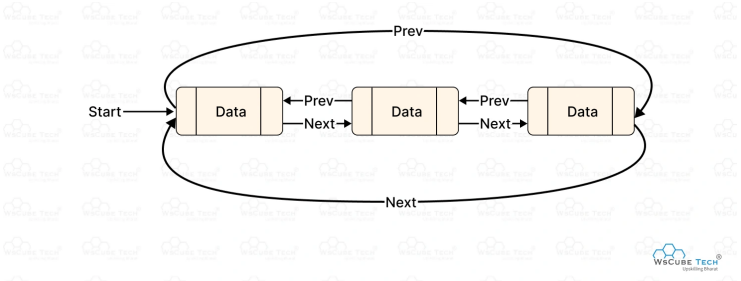

Figure 1: Visualization of Doubly Linked List Operations

—

3 User Interface (ui.c)

The user interface is handled in `ui.c`. This file contains functions that display the welcome banner and handle user input to determine whether to resume a previous session or start a new one.

3.1 Welcome Banner

The `printBanner` function displays a welcome banner when the program starts. This banner is designed to be visually appealing and informative.

```
void printBanner() {
  printf("      --------              --   ------  -                  --
        \n");
  printf("    /-  --/ /-  ---     / / / / / /-(-)--- ---  ---- -/ /
    ---- \n");
  printf("     / / / -- \\/ - \\   / / / / / --/ / -- `-- \\/ -- `/
    --/ - \\\n");
```

```
  printf("    / / / / / /  --/  / /-/ / / /-/ / / / / / / /-/ / /-/
      --/\n");
  printf("   /-/ /-/ /-/\\---/   \\----/-/\\--/-/-/ /-/ /-/\\--,-
    /\\--/\\---/ \n");
  printf("
      \n");
  printf("             -------                          -       --\n")
    ;
  printf("          / ----(_)---  ----  -----  -----(_)--- -/ /\n");
  printf("         / /-  / / -- \\/ -- '/ -- \\/ ---/ / -- '/ / \n"
    );
  printf("        /--/ / / / / / /-/ / / / / /--/ / /-/ / /  \n");
  printf("       /-/  /-/-/ /-/\\--,-/-/ /-/\\---/-/\\--,-/-/   \n"
    );
  printf("                                                \n");
  printf("         ------                 --              \n");
  printf("       /-  --/-------- ------/ /-----  -----\n");
  printf("        / / / ---/ -- '/ ---/ //-/ - \\/ ---/\n");
  printf("       / / / /  / /-/ / /--/ ,< /  --/ /    \n");
  printf("      /-/ /-/   \\--,-/\\---/-/|-|\\---/-/     \n");
}
```

3.2 Session Management

The `printWelcome` function prompts the user to decide whether to resume a previous session or start a new one. If the user chooses to start a new session, the program creates a new directory and file to store transactions.

```
bool printWelcome() {
  char choice[10];
  printBanner();
  printf("This tracker is built specifically to track income and
    expenses.\n");
  printf("\nWould you like to resume your previous session? (y/n):
    ");
  scanf("%c", choice);
  return printChoice(choice);
}
```

—

4 Main Program Logic (program.c)

The main logic of the program is handled in `program.c`. This file contains functions that process user commands, such as adding income or expenses, deleting transactions, and saving data to a file.

4.1 Key Functions in program.c

4.1.1 start()

The `start()` function initializes the program. It creates a sentinel node for the linked list, displays the welcome message, and loads previous transactions from

a file (if available). It then enters a loop to process user commands until the user decides to quit.

```
void start() {
  Node* sen = createNode(sen, "null", "null", 0, "null", sen);
  sen->size = 0;
  bool wantToContinue = printWelcome();

  if (wantToContinue) {
    readFile(sen); // Load transactions from file
    printf("Current Balance: %.2f\n", sen->balance);
    if (sen->balance < 0) {
      printf("Budget Status: !!! Over budget !!!\n");
    }
    else {
      printf("Budget Status: *** Within budget ***\n");
    }
  }

  printCommand(); // Display available commands
  bool finished = false;
  while (!finished) {
    finished = processCommand(sen); // Process user commands
    printf("\n");
  }
}
```

4.1.2 processCommand()

This function reads user input and determines which action to take based on the command entered. It uses the `validateCommand()` function to check the validity of the command and execute the corresponding action.

```
bool processCommand(Node* sen) {
  char command[30];
  printf("Enter command: ");
  scanf(" %[^\n]s", command); // Read user input
  return validateCommand(command, sen); // Validate and execute
    command
}
```

4.1.3 validateCommand()

This function validates the user's command and calls the appropriate function to handle it. The supported commands are: - **‘add income‘**: Adds an income transaction. - **‘add expense‘**: Adds an expense transaction. - **‘delete [position]‘**: Deletes a transaction at the specified position. - **‘print‘**: Displays all transactions and the current balance. - **‘quit‘**: Saves transactions to a file and exits the program.

```
bool validateCommand(char* command, Node* sen) {
  bool wantToQuit = false;
  bool isDelete = false;

```

```
  if (strlen(command) >= 6) {
    char sliceCommand[6] = "";
    slice(command, sliceCommand, 0, 6);
    if (strcmp(sliceCommand, "delete") == 0) {
      deleteList(sen, command); // Handle delete command
      isDelete = true;
    }
  }
  if (strcmp(command, "add income") == 0) {
    addIncome(sen); // Handle add income command
  }
  else if (strcmp(command, "add expense") == 0) {
    addExpense(sen); // Handle add expense command
  }
  else if (strcmp(command, "print") == 0) {
    displayList(sen); // Handle print command
  }
  else if (strcmp(command, "quit") == 0) {
    wantToQuit = quit(sen); // Handle quit command
  }
  else if (!isDelete) {
    printf("I don't know what you mean..."); // Invalid command
  }
  return wantToQuit;
}
```

4.1.4 addIncome() and addExpense()

These functions allow the user to add new income or expense transactions. They prompt the user for a description and amount, and then add the transaction to the linked list. The user can also choose to insert the transaction at a specific position in the list.

```
void addIncome(Node* sen) {
  char description[50];
  double income;
  char choice[10];
  printf("Enter income description: ");
  scanf(" %[^\n]s", description); // Read income description
  char type[30] = "INC"; // Set transaction type to "INC" (income)

  printf("Enter amount: ");
  scanf(" %lf", &income); // Read income amount
  getchar();

  printf("Do you wanna insert at your desired position? (y/n): ");
  scanf(" %c", choice); // Ask if user wants to insert at a
    specific position
  if (*choice == 'y' || *choice == 'Y') {
    insertPos(sen, type, description, income); // Insert at
    specific position
  }
  else {
    addLast(sen, type, description, income); // Add to the end of
    the list
    printf("Income added.");
```

```
  }

  // Update and display balance
  printf("\nCurrent Balance: %.2f\n", sen->balance);
  if (sen->balance < 0) {
    printf("Budget Status: !!! Over budget !!!\n");
  }
  else {
    printf("Budget Status: *** Within budget ***\n");
  }
}
```

4.1.5 deleteList()

This function handles the deletion of a transaction at a specific position. It marks the transaction for deletion by updating its status and adjusts the user's balance accordingly.

```
void deleteList(Node* sen, char* command) {
  char commandName[30];
  int position, ret;

  ret = sscanf(command, " %s %d", commandName, &position); // Parse
     position from command
  markDelete(sen, position); // Mark transaction for deletion
  printf("Transaction at position %d marked for deletion.",
    position);

  // Update and display balance
  printf("\nCurrent Balance: %.2f\n", sen->balance);
  if (sen->balance < 0) {
    printf("Budget Status: !!! Over budget !!!\n");
  }
  else {
    printf("Budget Status: *** Within budget ***\n");
  }
}
```

4.1.6 readFile()

This function reads transactions from a file (‘transactions.txt‘) and loads them into the linked list. It parses each line of the file to extract the transaction type, description, and amount, and then adds the transaction to the list.

```
void readFile(Node* sen) {
  FILE* file_ptr;
  file_ptr = fopen("./logs/transactions.txt", "r"); // Open file
    for reading
  if (file_ptr == NULL) {
    printf("File does not exist\n");
    return;
  }

  // Parse file and add transactions to the list
  while ((ch = fgetc(file_ptr)) != EOF) {
```

```
    if (ch == '\n') {
      amount = atof(amountStr);
      addLast(sen, type, description, amount); // Add transaction
    to list
      // Reset variables for the next transaction
      type[0] = '\0';
      description[0] = '\0';
      amountStr[0] = '\0';
    } else {
      // Parse transaction details
      if (!first) {
        strncat(type, &ch, 1);
        if (ch == '|') first = true;
      } else if (first && !second) {
        strncat(description, &ch, 1);
        if (ch == '|') second = true;
      } else if (first && second && !third) {
        strncat(amountStr, &ch, 1);
      }
    }
  }
  fclose(file_ptr); // Close the file
}
```

4.1.7 saveData()

This function saves all transactions to a file when the user chooses to quit the program. It writes each transaction's type, description, and amount to the file in a formatted manner.

```
void saveData(Node* sen) {
  Node* current = sen->next;
  FILE* file_ptr;
  file_ptr = fopen("./logs/transactions.txt", "w"); // Open file
    for writing
  if (file_ptr == NULL) {
    printf("File does not exist\n");
    return;
  }

  // Write transactions to file
  while (current != sen) {
    fprintf(file_ptr, "%s|%s|%.2f\n", current->type, current->
    description, current->amount);
    current = current->next;
  }
  fclose(file_ptr); // Close the file
}
```

4.1.8 quit()

This function handles the program's exit. It checks if there are any transactions to save and then calls `saveData()` to write the transactions to a file before exiting.

```
bool quit(Node* sen) {
  if (sen->size == 0) {
    printf("No transactions to save. Exiting program.\n");
    return true;
  }
  printf("Saving transactions to file...\n");
  update(sen); // Remove transactions marked for deletion
  saveData(sen); // Save remaining transactions to file
  printf("Done. Exiting program.\n");
  return true;
}
```

5 Main Function (main.c)

The `main.c` file is the entry point of the program. It calls the `start` function, which initializes the program and begins processing user commands.

```
int main() {
  start();
  return 0;
}
```

6 Conclusion

The Income and Expense Tracker program is a simple yet effective tool for managing personal finances. The use of a **doubly linked list** allows for efficient management of transactions, while the user interface provides a clear and intuitive way to interact with the program. The program's modular design makes it easy to extend and modify in the future.