System Skills Assignment 2

*Disclaimer: I'd like to mention that I wrote my C code in CLion, meaning that the code indents are kinda all over the place (it looks worse in Github) since I sometimes used their autofill to write certain sections of code. Reformatting would take too long, and CLion's builtin one won't work for my **main.c** for whatever reason.*

Linked List Implementations + Advantages

```C/C++
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

typedef struct { // define the structure for Node
       char* transaction;
       struct Node* next;
       int mark;
} Node;

struct Node* create(char* value); // create a new node via memory allocation
void insertFirst(struct Node** head, char* value, int mark); // insert at first
position
void insert(struct Node** head, char* value, int mark, int position); // insert
a node at some index
void insertLast(struct Node** head, char* value, int mark); // insert at last
position

void deleteFirst(struct Node** head); // delete the first node
void delete(struct Node** head, int position); // delete a node at some index
void deleteLast(struct Node** head); // delete the last node

#endif //LINKEDLIST_H
```

The Linked List that is essentially the heart of my code is loosely based on these definitions in the `LinkedList.h` header (shown above). It has all the essential functions that a linked list has (and admittedly is loosely structured based on GeeksForGeeks' singly linked list, although altered to meet assignment criteria).

The special `mark` variable is explained in my main.c, but I'll put it here for sake of readability.

```C/C++
  /*
       MARK basically stores what we'd do with each element at the index of the
       array.
       This is for displaying what to do with print, as well as after quit.
             -1 is to delete - will appear as --- d
             0 is do nothing - will appear as (saved)
             1 is to add at last - will appear as (new)
             2 is to add at position - will appear as +++ i
   */
```

The advantages of using a linked list in C as opposed to other data types (such as normal arrays) is that linked lists are dynamic, meaning it can grow or shrink based on whatever operations we do on it - a trait that is a must in our financial tracker. It is also more efficient in terms of runtime compared to normal arrays, but I won't go in more detail on this since it leans more towards the Data Structures class. The first point is probably the only point that matters in this implementation.

Linked List Code

```C/C++
struct Node* create(char* value) {
  struct Node* newNode = malloc(sizeof(struct Node));
  newNode->transaction = strdup(value);
  newNode->next = NULL;
  newNode->mark = 0; // do nothing with it yet
  return newNode;
}

void insertFirst(struct Node** head, char* value, int mark) {
  struct Node* newNode = create(value);
  newNode->next = *head;
  newNode->mark = mark;
  *head = newNode;
}

void insert(struct Node** head, char* value, int mark, int position) {
  struct Node* newNode = create(value);
  newNode->mark = mark;
```

```
  if (position == 0) {
       insertFirst(head, value, mark);
       return;
  }

  struct Node* temp = *head;
  int current = 0;
  while (temp != NULL && current < position - 1) { temp = temp->next; }

  if (temp == NULL) {
       printf("Invalid range. Try again.\n");
       return;
  }

  newNode->next = temp->next;
  temp->next = newNode;
}

void insertLast(struct Node** head, char* value, int mark) {
  struct Node* newNode = create(value);
  newNode->mark = mark;

  if (*head == NULL) { // if list is NULL or empty,
       *head = newNode;
       return;
  }

  struct Node* temp = *head;

  while (temp->next != NULL) {
       temp = temp->next;
  }
  temp->next = newNode;
}

void deleteFirst(struct Node** head) {
  if (*head == NULL) { // Check if the list is empty
       printf("List is empty, nothing to delete.\n");
       return;
  }

  struct Node* temp = *head;
  *head = temp->next;
  free(temp);
```

```
}

void delete(struct Node** head, int position) {
  if (*head == NULL) { // error handling for if the linked list is invalid
       printf("Invalid linked list.\n");
       return;
  }
  struct Node* temp = *head;

  if (position == 0) {
       *head = temp->next;
       free(temp);
       return;
  }

  int current = 0;
  while (temp != NULL && current < position - 1) { // traverse to the given
position
       temp = temp->next;
       current++;
  }
  if (temp == NULL || temp->next == NULL) {
       printf("Unable to delete node. Position may be invalid\n"); // error
message for index error
       return;
  }

  struct Node* next = temp->next->next; // remove the node at position + point
previous node to next node
  free(temp->next);
  temp->next = next;
}

void deleteLast(struct Node** head) {
  struct Node* temp = *head;
  if (temp->next == NULL) { // the only element in the linked list
       free(temp);
       *head = NULL;
       return;
  }
  while (temp->next->next != NULL) {
       temp = temp->next;
  }
  free(temp->next);
```

```
  temp->next = NULL;
}
```

What Happens When The Program Is First Run?

```C/C++
char answer[10];
  printf("Welcome to your Personal Finance Tracker!\n\nDo you want to resume a
previous session? (y/n): ");
  scanf("%s", answer);

  /*
       It'd be too much of an eyesore to implement case recognition for user
input, so
       I'd like to assume the input is lowercase.
   */

  struct Node* Transactions = NULL;

  if (strcmp(answer, "y") == 0) {

       printf("Resuming from last session...\n");
       FILE *file = fopen("./logs/transaction_log.txt", "a+");

       char line[256]; // buffer
       while (fgets(line, sizeof(line), file)) {
       insertLast(&Transactions, line, 0);
       }
       fclose(file); // done with file so we close it
       printf("Previous transactions loaded.\n");

       getTotal(&Transactions, 0);

  } else if (strcmp(answer, "n") == 0) {

       printf("Created a new transaction log. You may begin to run
commands.\n");

  } else {

       printf("Invalid input. Please restart the program.\n");
```

```
        return 0; // easier to just restart program than make them do a lot of
inputs

    }
```

The following code is inside the `main()` function that is run when we execute `./fintrack`

- If we answer “n” as in no, we do not want to read from an existing file, the program will just use the new Transactions linked list we created as a basis for whatever operations we are gonna do next.

```
lilconx@mik-laptop:~/Desktop/C Programs/a2/a02-c-financial-tracker-mik5plays$ ./release/fintrack
Welcome to your Personal Finance Tracker!

Do you want to resume a previous session? (y/n): n
Created a new transaction log. You may begin to run commands.

Enter command:
```

- On the contrary, if we answer “y” as in yes, we want to read from an existing file, the program reads from “logs/transaction_log.txt” if it exists. If it doesn’t, in theory it just creates a new transaction_log.txt file and rolls with the empty linked list Transactions we already created.

Assume our text file looks like this:

```
Unset
INC|Salary|100.00
INC|Allowance|1500.50
EXP|Doom|-10.00
INC|Integer|222.00
```

Doing “y” will give this sample output:

```
lilconx@mik-laptop:~/Desktop/C Programs/a2/a02-c-financial-tracker-mik5plays$ ./release/fintrack
Welcome to your Personal Finance Tracker!

Do you want to resume a previous session? (y/n): y
Resuming from last session...
Previous transactions loaded.
Current Balance: $1812.50
Budget Status: Within Budget

Enter command:
```

Commands

command loop

C/C++

```
  int hasQuit = 0;
// checker for if the user has quit the program, so we terminate the while loop

  while (getchar() != '\n'); // clear input buffer

  while (hasQuit == 0) { // gonna keep looping commands until user says quit

        printf("\nEnter command: ");

        char input[50]; // buffer of 50 characters, that we should not exceed
        if (fgets(input, sizeof(input), stdin) != NULL) { // do some error
handling
        ssize_t len = strlen(input);
        if (len > 0 && input[len - 1] == '\n') {
               input[len - 1] = '\0';  // remove newline, if exists
        }
        }

        // for readability i got rid of the stuff inside each command. this is
just for explanation purposes in the summary.
        if (strcmp(input, "add income") == 0) {

        } else if (strcmp(input, "add expense") == 0) {

        } else if (strcmp(input, "delete") == 0) {

        } else if (strcmp(input, "print") == 0) {

        } else if (strcmp(input, "quit") == 0) {

        } else {

        printf("\nInvalid command! Please try again.\n");

        }
  }
  printf("Done. Exiting program.\n");
```

```
  free(Transactions); // we have finished our work. so we free memory and
finish the program.
  return 0;
```

For the commands, I decided to implement a “loop” structure where you’ll just keep entering commands until `quit` is entered, where we do the stuff that happens after we quit + end the program.

I used `scanf` and `fscanf` so the *clear input buffers* functions were necessary. Without them, it’d mess up the command input for other commands. The rest of the inputs I used `fgets` and similar, as this was a better way.

For most of my inputs, I copy-pasted the same buffer/new-line checker that truncates the input so that it doesn’t destroy my code.

add income, add expense, print

The following commands are dependent on the two functions that I wrote situated outside of `main()` that I will explain shortly after:

```
C/C++
if (strcmp(input, "add income") == 0) {

       add_transaction(&Transactions, "income");

} else if (strcmp(input, "add expense") == 0) {

       add_transaction(&Transactions, "expense");

} else if (strcmp(input, "print") == 0) {

       getTotal(&Transactions, 1);

}
```

For modularity, both `add income` and `add expense` actually use the same function, just slightly different parameters. In short, it just asks for the description (name), position to insert in (-1 if we just want to append), and the amount (strictly positive). It forms the single string that is stored inside the linked list in the `transaction` variable.

```C/C++
void add_transaction(struct Node** transactions, char* INC_OR_EXP) {
	printf("Enter %s description: ", INC_OR_EXP);
	char DESCRIPTION[50]; // buffer of 50 characters, that we should not exceed

	if (fgets(DESCRIPTION, sizeof(DESCRIPTION), stdin) != NULL) { // do some error handling
	ssize_t len = strlen(DESCRIPTION);
	if (len > 0 && DESCRIPTION[len - 1] == '\n') {
		DESCRIPTION[len - 1] = '\0';  // remove newline, if exists
	}
	}

	printf("Enter position of %s. Enter -1 to append to the last position\nNote that 0 is the first position for this: ", INC_OR_EXP);
	int position;
	if (fscanf(stdin, "%d", &position) != 1) {
	printf("Invalid input. Please try again.\n");
	return;
	}

	while (getchar() != '\n'); // clear input buffer for scanf type stuff

	printf("Enter amount of %s: ", INC_OR_EXP);
	char AMOUNT_AS_STR[50];
	char* STR_PTR; // pointer for error handling.

	if (fgets(AMOUNT_AS_STR, sizeof(AMOUNT_AS_STR), stdin) != NULL) { // do some error handling
	ssize_t len = strlen(AMOUNT_AS_STR);
	if (len > 0 && AMOUNT_AS_STR[len - 1] == '\n') {
		AMOUNT_AS_STR[len - 1] = '\0';  // remove newline, if exists
	}
	}

	float AMOUNT = strtof(AMOUNT_AS_STR, &STR_PTR);
```

```
        if (*STR_PTR != '\0') {
        printf("Rejected. Invalid amount.\n");
        return;
        }

        if (AMOUNT < 0) { // Reject negative values.
        printf("Rejected. Amount must be greater than 0.\n");
        return;
        }

        char result[256]; // create our entry that we will put inside the linked
list.
        result[0] = '\0';
        strcat(result, "INC|");
        strcat(result, DESCRIPTION);
        strcat(result, "|");
        if (strcmp(INC_OR_EXP, "expense") == 0) {
        strcat(result, "-"); // for an expense, we include the negative sign.
        }
        strcat(result, AMOUNT_AS_STR);
        if (strchr(AMOUNT_AS_STR, '.') == NULL) { // add .00 manually if it's an
integer
                strcat(result, ".00");
        }

        if (position == -1) {
        insertLast(transactions, result, 1); // append to linked list, mark as an
append
        } else {
        insert(transactions, result, 2, position); // append to linked list, mark
as insert at
        }

        printf("\n");
        getTotal(transactions, 0); // run the total to see whether we're still on
budget or not.

}
```

Also for modularity, I made another function that's solely responsible for outputting the transaction information in the terminal. This is because print, as well as some other happenings (when you first run the program, after adding incomes and expenses, etc.) use the same code. `LIST_TRANSACTIONS` is exclusive for when I use print, as that's when I want to print out the linked list and stuff.

C/C++

```
void getTotal(struct Node** transactions, int LIST_TRANSACTIONS) {
  if (*transactions == NULL) { return; } // error handling
  struct Node* current = *transactions;
  float TOTAL = 0;
  int CURRENT = 0;

  if (LIST_TRANSACTIONS == 1) {
       printf("\n[ Transactions ]\n");
  }

  while (current != NULL) {
       char* temp = strdup(current->transaction);

       char* token = strtok(temp, "|"); // split value entry into three parts
based on the delimiter '|'
       char EXP_OR_INC[100];
       strcpy(EXP_OR_INC, token);

       token = strtok(NULL, "|");
       char NAME[256];
       strcpy(NAME, token);

       token = strtok(NULL, "|");
       float AMOUNT = atof(token);

       TOTAL += AMOUNT; // add to total costs

       if (LIST_TRANSACTIONS == 1) { // special case to list all the
transactions in print
       printf("%-20d", CURRENT);
       printf("%-20s ", NAME);
       printf("%-20.2f ", AMOUNT);
       if (current->mark == -1) {
              printf("%s", "--- d\n");
       } else if (current->mark == 0) {
              printf("%s", "(saved)\n");
       } else if (current->mark == 1) {
              printf("%s", "(new)\n");
       } else if (current->mark == 2) {
              printf("%s", "+++ i\n");
       }
       }

       current = current->next;
```

```
        CURRENT++;

    }

    printf("Current Balance: $%.2f\n", TOTAL); // round to two decimal places

    if (TOTAL > 0) {
        printf("Budget Status: Within Budget\n");
    } else {
        printf("Budget Status: Over Budget !\n");
    }

}
```

delete

C/C++

```
        } else if (strcmp(input, "delete") == 0) {

        printf("Delete position: ");
        int position;

        if (fscanf(stdin, "%d", &position) != 1) {
                printf("Invalid input. Please try again.\n");
                continue;
        } else {
                struct Node* temp = Transactions;
                for (int i = 0; i < position; i++) {
                temp = temp->next;
                }
                if (temp == NULL) {
                printf("Invalid position. Please try again.\n");
                while (getchar() != '\n'); // clear input buffer
                continue;
                }

                temp->mark = -1; // marked for deletion, but we don't delete just
yet.

                printf("Transaction at position %d is marked for deletion.\n",
position);
```

```
                while (getchar() != '\n'); // clear input buffer
        }

        }
```

The delete command asks for a position **(important to stress the fact that I decided to use 0 as the first position, rather than 1 in the sample output. I am just more used to that style when working with other languages)** and then marks said position for deletion. Some error checking stuff then and there, but apart from that it's essentially just what I said.

quit

```
C/C++
        } else if (strcmp(input, "quit") == 0) {

        FILE *file = fopen("./logs/transaction_log.txt", "w"); // we are
overwriting the file.

        if (file == NULL) {
                printf("Unable to open file. Program must be restarted!\n");
                return 0;
        }

        struct Node* TRANSACTION_PTR = Transactions;
        while (TRANSACTION_PTR != NULL) {
                if (TRANSACTION_PTR->mark != -1) {
                char *string = TRANSACTION_PTR->transaction;
                ssize_t len = strlen(string);
                if (len > 0 && string[len - 1] == '\n') { // get rid of newlines
(then add my own to prevent confusion)
                        string[len - 1] = '\0';
                }
                if (TRANSACTION_PTR->next != NULL) { fprintf(file, "%s\n",
string); }
                else { fprintf(file, "%s", string); }
                }
                TRANSACTION_PTR = TRANSACTION_PTR->next;
        }
        fclose(file); // done with file so we close it
```

```
        printf("Saving transactions to file...\n");
        hasQuit = 1; // we have quit.

        }
```

The quit command will write our linked list to specifically `./logs/transaction_log.txt`. Note that it will overwrite whatever's inside since I'm using write mode. Some error checking + prevention method that involves getting rid of messy newlines that might screw up the text file. But apart from that, there's not a lot to explain.

After doing that, `hasQuit` changes to 1 meaning the while loop gets terminated and the program exits.

Sample Code Output

This is what transaction_log.txt looks like before (basically the same as the previous screenshot).

```
Unset
INC|Salary|100.00
INC|Allowance|1500.50
EXP|Doom|-10.00
INC|Integer|222.00
```

Using `add income` to **append** a new entry that looks like "`INC|Won A Bet|100.00`"

```
Unset
Enter command: add income
Enter income description: Won A Bet
Enter position of income. Enter -1 to append to the last position
Note that 0 is the first position for this: -1
Enter amount of income: 100

Current Balance: $1912.50
Budget Status: Within Budget

Enter command: print
```

```
[ Transactions ]
0                   Salary              100.00              (saved)
1                   Allowance           1500.50             (saved)
2                   Doom                -10.00              (saved)
3                   Integer             222.00              (saved)
4                   Won A Bet           100.00              (new)
Current Balance: $1912.50
Budget Status: Within Budget
```

Using `add expense` to **add a new entry at index 2** that looks like "`EXP|Groceries|-45.00`"

Unset

```
Enter command: add expense
Enter expense description: Groceries
Enter position of expense. Enter -1 to append to the last position
Note that 0 is the first position for this: 2
Enter amount of expense: 45

Current Balance: $1867.50
Budget Status: Within Budget

Enter command: print

[ Transactions ]
0                   Salary              100.00              (saved)
1                   Allowance           1500.50             (saved)
2                   Groceries           -45.00              +++ i
3                   Doom                -10.00              (saved)
4                   Integer             222.00              (saved)
5                   Won A Bet           100.00              (new)
Current Balance: $1867.50
Budget Status: Within Budget
```

Using `delete` to **delete “Doom” that is stored at index 3.**

```
Unset
Enter command: delete
Delete position: 3
Transaction at position 3 is marked for deletion.

Enter command: print

[ Transactions ]
0                   Salary              100.00              (saved)
1                   Allowance           1500.50             (saved)
2                   Groceries           -45.00              +++ i
3                   Doom                -10.00              --- d
4                   Integer             222.00              (saved)
5                   Won A Bet           100.00              (new)
Current Balance: $1867.50
Budget Status: Within Budget
```

Quitting and saving everything.

```
Unset
Enter command: quit
Saving transactions to file...
Done. Exiting program.
```

This is what `transaction_log.txt` looks like after all that:

```
Unset
INC|Salary|100.00
INC|Allowance|1500.50
INC|Groceries|-45.00
INC|Integer|222.00
INC|Won A Bet|100.00
```