CSCB09H3: Software Tools and Systems Programming
Assignment 2: Buxfer

Due Date: **11:59 pm Sunday, March 3, 2019**
Worth: 10% of your final grade.

Introduction:

Buxfer is a service that lets groups of people track shared expenses. For example, roommates might want to track shared expenses such as rent, utilities, and groceries, or colleagues might want to keep track of shared lunch bills. For each expense, Buxfer records the person who paid the expense and the expense cost. It allows group members to examine the amount that individuals have paid, look at the history of all transactions, or determine the group member that is currently owing the most. Buxfer started as a small C program written by three graduate students at Carnegie Mellon University for their personal use and has since grown into a full-blown company (https://www.buxfer.com/). Your task in this assignment is to write a basic version of Buxfer.

Data structures

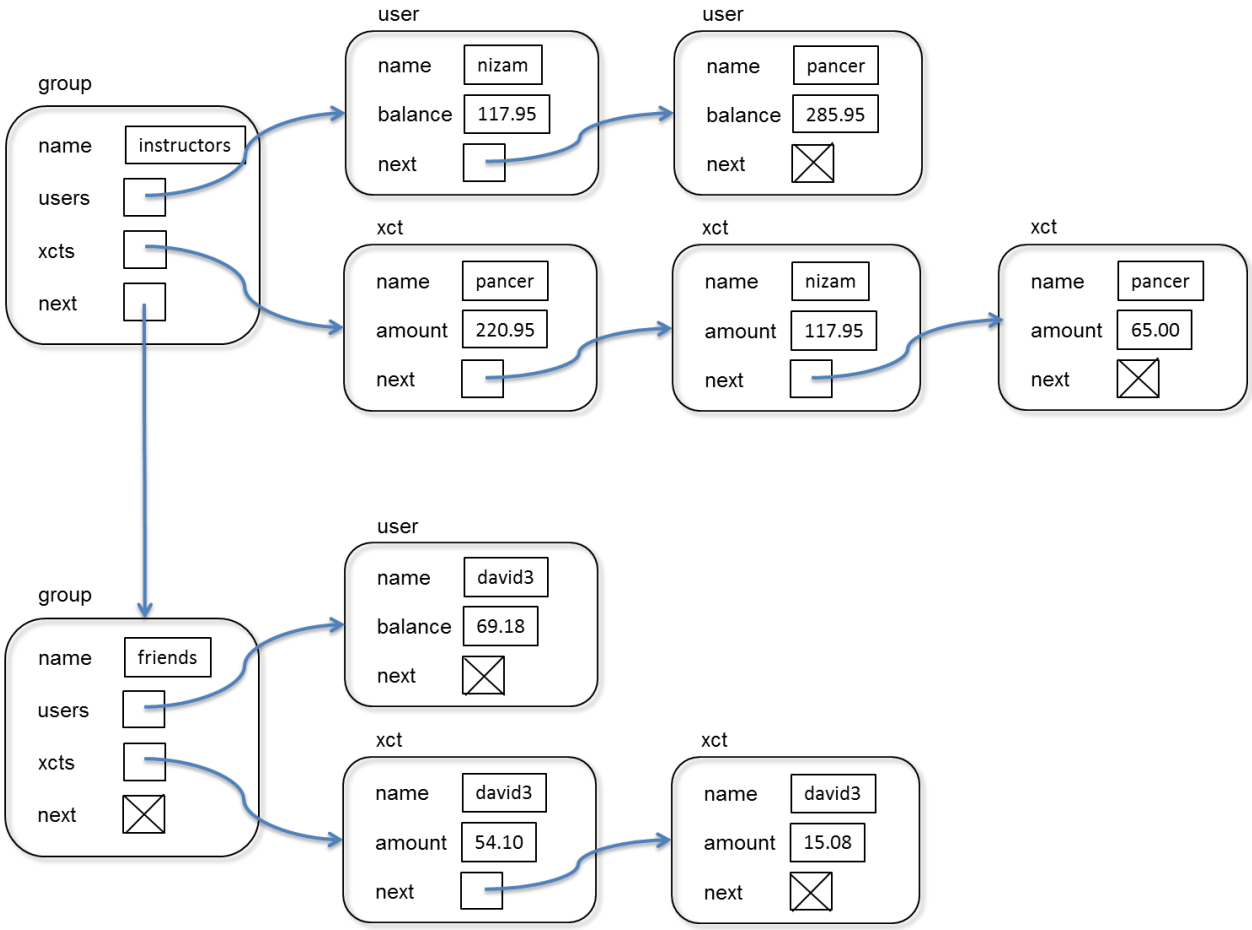
A Buxfer application can manage multiple groups, and for each group you must keep track of the group's members and the group's transactions (expenses paid by users). This means that we need three different data structures: one to keep track of the groups, one to keep track of the users of a group, and one for a group's transactions. Since we don't know the number of groups that will be created, the users that will join a group, or the number of transactions that will be posted, we will use linked lists to store groups, users and transactions. The picture on the next page shows what these data structures look like for an example with two different groups, one with two users and three transactions, and the other with one user and two transactions.

A node in the linked list of groups (the list arranged vertically on the left in the picture) stores the name of the group, a pointer to the first node of the linked list that stores the users of this group, a pointer to the first node of the linked list that stores all the transactions posted for this group, and a pointer to the next group list node.

A user list node keeps track of the relevant information for this user, i.e. his/her name, balance (the total amount of expenses he/she has paid so far) and a pointer to the next element of the list of users.

A transaction (xct) list node keeps track of all the information about a transaction, i.e. the name of the user who paid for it, the amount, and a pointer to the next transaction list node.

The user lists are sorted by the user's balances, with the user with the lowest balance first in the list and the user with the highest balance last in the list. This makes it easier to find the user who has paid the least so far. The transaction lists are sorted by the time when a transaction was added, with the most recent transaction first in the list. This makes it easy to output a sorted history of the transactions in a group.



Operations supported by Buxfer

A Buxfer application supports the following user commands.

- `add_group group_name`: Register a new group with name `group_name`.
- `list_groups`: List the names of all groups that are currently registered.
- `add_user group_name user_name`: Register a new user with name `user_name` as a member of group `group_name`.
- `list_users group_name`: List the names of all users of group `group_name` together with their current balances (i.e. how much each has paid so far), sorted by their balances (lowest payer first).
- `user_balance group_name user_name`: Return the balance of user `user_name` in group `group_name`
- `remove_user group_name user_name`: Remove user `user_name` from group `group_name`
- `underpaid group_name`: Output the name of the user in group `group_name` who has paid the least.
- `add_xct group_name user_name amount`: Add a new transaction for group `group_name`. The transaction is paid by `user_name` and the amount is `amount`.
- `recent_xct group_name num_xct`: List the `num_xct` most recent transactions for group `group_name`.
- `quit`: Shut down buxfer

The buxfer executable can be started from the commandline with either zero or one argument as follows:

```
./buxfer [filename]
```

If buxfer is run without any arguments, it starts in interactive mode, i.e. it will display a prompt and wait for the above commands at the command line. If run with one argument, it expects this argument to be the name of a file that contains one buxfer command per line (the commands are those from the list above) and will execute those commands from the file.

Starter code

Since this is your first C assignment, we are providing you with some starter code. The starter code can be found here:

```
/courses/courses/cscb09w19/nizamnau/a2/
```

This directory contains a `Makefile` and three files called `buxfer.c`, `lists.h`, and `lists.c` that provide the implementation of a skeleton of the buxfer data structures. `buxfer.c` provides the code to read and parse the user commands described above, and then calls the corresponding C function that implements the functionality of a user command. When run without command line arguments, buxfer will wait for commands from the console (standard input); when run with a filename as a command line argument, it will read commands from the specified input file. Read the starter code carefully to familiarize yourself with how it works.

Function prototypes for all functions implementing the various user commands are provided in `lists.h`. So that the starter code compiles without errors, these functions are currently implemented in `lists.c` as stubs (i.e. functions whose body is empty except for the return statement), In addition, there are prototypes and stub implementations for a number of helper functions that you will use to implement the user command functions. Your task will be to complete the implementation of all these functions.

We have also provided a sample input file `batch_commands.txt` that contains a series of buxfer user commands that buxfer will execute in batch mode if run with the name of the file as a command line argument.

Your tasks

You will complete the functions defined in `lists.c`. The comment above each function stub further explains the behaviour of the function and sometimes gives some implementation tips.

Note that you are not allowed to modify `buxfer.c` or `lists.h`. The only C file you are modifying is `lists.c`. You may add new functions to `lists.c` but do not change the names or prototypes of the functions we have given you.

Part I: Managing groups (20%)

Start by building a solution that only manages the groups. i.e. implement the `add_group` and `list_groups` commands. This will involve completing the following C functions.

- `int add_group(Group **group_list, const char *group_name)`
- `void list_groups(Group *group_list)`
- `Group *find_group(Group *group_list, const char *group_name)`

Part II: Managing users (40%)

Next, write the code for the functions that are called for the user commands `add_user`, `remove_user`, `list_users`, `user_balance`, and `under_paid`.

- `int add_user(Group *group, const char *user_name)`
- `int remove_user(Group *group, const char *user_name)`
- `int list_users(Group *group)`
- `int user_balance(Group *group, const char *user_name)`
- `int under_paid(Group *group)`

Your code for the `add_user`, `remove_user`, and `user_balance` functions above should use the following helper function, which you also need to implement (again, a stub is provided):

- `User *find_prev_user(Group *group, const char *user_name)`

Part III: Managing transactions (40%)

Write the following two functions that implement the functionality for the user commands `add_xct` and `recent_xcts`.

- `int add_xct(Group *group, const char *user_name, double amount)`
- `int recent_xct(Group *group, long num_xct)`

At this point, you will also implement the following helper function and add calls to it in your `remove_user` function.

- `void remove_xct(Group *group, const char *user_name)`

Error handling

The comments for the functions you are to implement define nearly all the error conditions you might encounter and tell you how to handle them. The main additional expectation is that you check the return value from `malloc` and terminate the program if `malloc` fails.

What to hand in

Commit to your repository in the `a2` directory all of the files that are needed to compile `buxfer.c` and run `buxfer`. A `Makefile` has been provided. **The markers must be able to run `make` such that buxfer is compiled with no warnings.** You may need to modify the `Makefile` to add more source code files.

Coding style and comments are important. Use good variable names, appropriate functions, descriptive comments, and blank lines. Remember that someone needs to read your code.

Please remember that if you submit code that does not compile, it will receive a grade of 0. The best way to avoid this potential problem is to write your code incrementally. For example, the starter code compiles and solves one small piece of the problem. Get a small piece working, commit it, and then move on to the next piece. This is a much better approach than writing a whole bunch of code and then spending a lot of time debugging it step by step.