--Lab 6--
Linked Lists and Hashtables
CS 24000
Introduction to C Programming
Due Date: April 4th 11:59pm

Introduction

In lab 6, you will implement both a doubly linked list, and a hash table. Using your linked list and hashtable implementation, you will design a simple application that processes logs of financial transactions. While working on this lab, both GDB and Valgrind will be useful tools to debug your code and diagnose memory errors.

Preparation

Type the following commands to get the tarball that will contain the initial code:

```
cd ~/cs240
tar -xvf /homes/cs240/2016Spring/Lab6/lab6-src.tar.gz
```

Batteries (not) included

The skeleton you are provided contains:

- Header files (**linkedlist.h**, **hashtable.h**, and **io.h**) declaring the functions you will implement
- A **hash.o** file containing the hash function for the hash table

For this lab, your responsibilities are to:

- Implement functions in **linkedlist.c**, **hashtable.c**, and **io.c** according to the descriptions in the corresponding header files. Do **NOT** write a main() function in any of these files.
- Implement a **main.c** file that contains the main() function for the transaction processing application (task 4)
- Implement your own test cases for both the final application and the individual data structure and processing function implementations. You should create additional *.c files for this purpose, but your implementation should not depend on these additional *.c testing files. We will **only** take your **linkedlist.c**, **hashtable.c**, **io.c** and **main.c** and run our own test cases against them.
- Create your own Makefile to compile and test your application and data structure and processing function implementation.

Header Files

For this lab you are only provided a set of header files (**linkedlist.h**, **hashtable.h**, and **io.h**). These header files represent the Application Programming Interface (API) that you are implementing. The implementation of the API should be done in the corresponding *.c files, so all the hash table functions should be in **hashtable.c**, etc.

You should use '#include "filename"' to include structure and function declarations. So for example, your **hashtable.c** file should contain a line at the top containing **#include "hashtable.h"** to include the hash table structure definitions and function declarations from **hashtable.h**.

WARNING: Do **NOT** modify the header files. These files contain an API we expect you to implement, this API includes the structure definitions. We will replace the headers during grading. If you make any changes to the headers, your code **WILL LIKELY** fail to compile during grading, resulting in a zero. The structures have all the fields you need to correctly implement the lab, do not add or remove and fields from the structure, nor should you change any of the names of the structure fields.

Task 0. Memory Leaks and Understanding Valgrind

Every task in this lab will be required to be memory leak free. As discussed in lecture, memory leaks occur when a program allocates memory but fails to later free that memory after it is no longer in use. When using dynamic data structures such as linked lists or hash tables, it is crucial that when the data structure is no longer in use, all memory allocated for the data structure is freed successfully (Note that dynamic data structures are built from many separate pieces of allocated memory, so all the different allocated pieces of memory must be freed). Otherwise, as an application creates and destroys dynamic data structures, memory will be leaked and eventually the application will run out of usable memory.

Fortunately, there are tools which exist to help you identify when and where a memory leak is happening. We have included a tutorial on Valgrind, a tool which will help you identify memory leaks and fix them. It is very important that you understand how to use this tool to minimize headaches trying to find out where your memory leaks are located.

Task 1. Implement a Linked List

For task 1 you will implement a doubly linked list. In **linkedlist.c** you will implement the set of functions described in **linkedlist.h:**

- linkedlist_t* create_linkedlist()
- void free_linkedlist(linkedlist_t*)
- int add_node(linkedlist_t*, node_t*)
- int remove_node(linkedlist_t*, node_t*)
- node_t* create_node(long int, short, const char*, const char*, double)
- void delete_node(node_t*)
- node_t* find_node(linkedlist_t*, long int)

A detailed description of each of these functions can be found in **linkedlist.h.**

Testing

You should write your own tests for your linked list. We recommend creating a file named **test_linkedlist.c** that contains a main() function that exercises your linked list implementation.You can then build a binary **test_linkedlist** from **test_linkedlist.c** and **linkedlist.c** that when run can test if your linked list is correctly implemented according to the specification. These are called **unit tests**. In the corporate world, for each function you write, you will have a set of tests to ensure your function is working correctly and according to the specifications given to you. By having one test file, you can easily add, modify, or delete tests which you think **thoroughly** test your function. This part is not graded. Make it as modular and helpful as possible to reduce future headaches!

It is highly recommended to have completed this task by March 28th

Task 2. Implement a Hash Table

For task 2, you will implement a hash table. You must use chaining to handle collision. The ‘table’ field in the hashtable structure is a pointer to an array of ‘**hashtable_ent_t**’ structure pointers. Each slot in the **hashtable_t** ‘**table**’ array is a pointer to the first node in a chain of zero or more **hashtable_ent_t** structures that hold the key-value pairs. We will verify that you implement chaining. In **hashtable.c** you will implement the set of functions described in **hashtable.h:**

- hashtable_t* create_hashtable(int)
- void free_hashtable()
- int get(hashtable_t*, const char*, double*)
- int set(hashtable_t*, const char*, double)
- int key_exists(hashtable_t*, const char*)
- int remove_key(hashtable_t*, const char*)

A detailed description of each of these functions can be found in **hashtable.h**.

Hash Function

Your hash table will use a hash function to determine the slot in the hash table in which to store a key-value pair. You must store key-value pairs in the array entry **table[hash(key) % table_len]**. You will use the **hash()** function declared in **hashtable.h** and provided to you in compiled form in **hash.o**. To include the **hash()** function in your program, **hashtable.h** must be #include'ed in your source code files, and you must add **hash.o** to the list of *.c files being compiled using gcc.

Chaining

Since multiple key may be mapped to the same hash value, there must be a way of handling these so called 'collisions' in the hash table. For this lab you must use 'chaining', where each slot in the hash table isn't a single key-value pair, but instead is a linked list, containing zero or more key-value pairs. The **table** field in the hash table structure is a pointer to an array of pointers. If there are not key-value pairs in a given slot, that pointer at **table[hash(key) % table_len]** must be NULL to indicate that slot is empty. Otherwise, the pointer value points to the first element of a linked list of one or more **hashtable_ent** structures allocated on the heap. For example, if two key-value pairs map to the same slot in the table, then **table[hash(key) % table_len]** would contain a pointer to a **hashtable_ent** value allocated from the heap containing the first key and value, while the **next** field in that **hashtable_ent** structure would contain a pointer to a second **hashtable_ent** structure containing the second key-value pair. Finally, the second **hashtable_ent** structure would contain a NULL pointer indicating that there are no following values.

Testing

You should write your own tests for your hashtable. We recommend creating a file named **test_hashtable.c** that contains a main() function that exercises your hash table implementation.You can then build a binary **test_hashtable** from **test_hashtable.c**, **hashtable.c** and **hash.o** that when run can test if your hash table is correctly implemented according to the specification.

It it highly recommended to have completed this task by April 1st

Task 3. Implement transaction processing functions

Now you will implement an application that will process transaction logs. Imagine you are a large bank that processes financial transactions between different companies. Each time two companies perform a money transfer, a new company adds an account to the bank, or a company leaves the bank, a new transaction is added to a file containing a log of the day's transactions. Additionally, the bank maintains an account balance file which contains pairs of companies and balances.

Data Structures

Your processing application will work with two different data structures: A **linkedlist_t** representing a linked list of all the transactions in the file (in order based on timestamp, lowest at the head to highest at the tail), and a **hashtable_t** representing the mappings from the name of a company, to the balance of that company's account.

Account File Functions

You will implement two functions that manipulate account files:

```
hashtable_t *read_accounts(FILE *fd); // Create a hash table mapping the company to balance
void write_accounts(FILE *fd, hashtable_t *accounts); // Write a hash table to fd
```

Account File Format

The format each line of the account balance file is as follows:

```
<CompanyName1> <Company1AccountBalance>
```

Example:

```
Pepsi 1000
Google 30000
Microsoft 34000
Facebook 31000
```

There may be leading or trailing whitespace characters (' ', '\t') at the beginning or end of the line, but only one space between the company name and the initial account balance.

Company Names

Each company name in both the account file and transaction log file will be no longer than 256 characters. It will consist of only alphanumeric characters and underscores '_'. Company names are **case sensitive**, so 'Microsoft' and 'microsoft' are considered different company names.

Transaction Log Functions

You will implement two functions that manipulate transaction logs:

- linkedlist_t *read_transactions(FILE* fd)
- int apply_transactions(linkedlist_t *transactions, hashtable_t *accounts)

Transaction Log List

You will create a linked list containing nodes representing transactions from a transaction log file. The nodes will be ordered by the transaction timestamp, with the smallest timestamp at the head of the linked list, and the largest timestamp (newest transaction) at the tail of the linked list. You will implement a function

```
linkedlist_t *read_transactions(FILE *fd)
```

that will

1. Create a new **linkedlist_t** using the appropriate function from **linkedlist.h**
2. Iterate through the file parsing each line representing a transaction
3. Create a node representing each transaction
4. Insert each node into the linked list in order by timestamp (lowest timestamps at the head, highest at the tail)

Transaction Log Format

Each transaction log is a text file containing lines representing financial transactions between companies. Below is a sample of the generic format of a line in the transaction log file representing a transaction:

```
<TIMESTAMP> <TRANSACTION_TYPE> <PARAM1> (... Further params)
```

As with the account file format, there may be tailing or leading spaces, but there will always be a single space between the different transaction parameters.

Timestamps

Every transaction starts with a positive integer timestamp in ASCII text indicating when the transaction occurred. A negative value is illegal, and read_transactions() should report an **error**

in that case by returning **NULL**. For each transaction, the timestamps of all following transactions must be greater in value. So if a transaction has timestamp 34387054, but the next transaction has timestamp 34387000, that is an **error**, while 34387055 or 34387056 are both valid timestamps since they are equal or greater than 34387054.

Transaction Types

Following the timestamp is the transaction type in string form. In the **io.h** file there are three macros (**ADD**, **REMOVE**, and **TRANSFER**). When constructing a node for the transaction list, you must use the corresponding integer code for the transaction in place of the string. For example, if a transaction line contains the string “ADD”, then you would store the integer 1 in the ‘transaction_type’ field in the linked list node. There are 3 types of transactions in the log:

Type	**Parameter 1**	**Parameter 2**	**Parameter 3**	**Example**
TRANSFER	company1	company2	amount	‘100 TRANSFER Microsoft Google 5’
ADD	company1	amount	<Not used>	‘102 ADD Pepsi 1000.5’
REMOVE	company1	<Not used>	<Not used>	‘104 REMOVE Facebook’

TRANSFER Transaction

The TRANSFER transaction has three parameters (company1, company2, and amount), each separated by spaces from one another and the transaction type. The TRANSFER transaction takes ‘amount’ dollars from the account of ‘company1’ in the accounts hash table, and adds it to the balance of ‘company2’ in the accounts hash table. The ‘amount’ field in the struct should always be greater than zero and the balance of a company in the ‘accounts’ hash table should never be become less than zero after a transaction. If either of these conditions occur, ‘apply_transactions()’ should return an error value. Ignore any transfer operations where both the sending and receiving company are the same. If either ‘company1’ or ‘company2’ do not exist in the hash table, that is also an error and ‘apply_transactions()’ should handle it by returning an error. An error is handled by returning -1.

ADD Transaction

The ADD transaction has two parameters (company1, amount). The ADD transaction inserts a new company with the name in the ‘company1’ field in the node struct into the ‘accounts’ hash table with a value of the ‘amount’ field in the node struct, representing the initial balance of that company.

REMOVE Transaction

The REMOVE transaction has one parameter (company1). The REMOVE transaction removes ‘company1’ from the accounts hashtable. If the company is not currently in the table, ‘apply_transactions()’ should return an error value.

Task 4. Implement a transaction processing application

Using the functions implemented in Step 3, write a file **main.c** that contains a main() function which will run the program. The program will take 3 parameters on the command line in the form:

```
./process <transaction_file> <account_in_file> <account_out_file>
```

Parameter Descriptions:

1. **<transaction_file>** will be the path to the file containing the list of transactions to process
2. **<account_in_file>** will be the path to the file containing the current account balance before the transactions are applied
3. **<account_out_file>** will be the path to the file which will contain the final account balances from the accounts hashtable

Your program should read in the transactions list from the file named by the **<transaction_file>** argument and the accounts hash table from the file named by the **<account_in_file>** argument. You should apply the transactions list against the accounts hash table and then write the output to the file named by the **<account_out_file>** argument.

Your main() function should check that all command line arguments are present. You should then open the corresponding files, and call the appropriate functions from io.c to perform IO and processing. If all of the functions execute successfully (returning non-NULL pointers or 0), then return 0 from main(). Otherwise, if an error occurred while calling any of the functions, then return 1 from main(). If too few or too many parameters are passed on the command line, then print

```
“Usage: <transaction_file> <account_in_file> <account_out_file>”
```

and return 1 from main().

Turnin

You should turnin the following files:

- **linkedlist.c**
- **hashtable.c**
- **io.c**
- **main.c**

We will replace the **test_*.c** and ***.h** header files during testing.

To turnin the project, run

turnin -c cs240 -p lab6 lab6-src/

and to verify your submission, run

turnin -v -c cs240 -p lab6

Grading Rubric

Task	**Weight**
Task 1 (Linked list)	25%
Task 2 (Hash Table)	25%
Task 3 (Processing functions)	25%
Task 4 (Complete Program)	25%
Total	**100%**

This project is due by April 4th at 11:59pm