Computer Science I Program 6: Theater Inventory (Hash Tables)

Please Check Webcourses for the Due Date

Please read the whole assignment before you start coding

Objective

Practice implementing a hash table via separate chaining hashing.

Background Story

Our theater is thriving due to the new loyalty program devised by employee Davin Hamter!

The theater now wants to improve its tracking of inventory. The theater sells many items. It must buy these items from suppliers at a wholesale price. For example, it may buy 1000 servings of popcorn for $300. Then, it charges customers a fixed price per item (say $6 per serving of popcorn).

For the purposes of this problem, assume that the theater starts with $100,000 to help buy supplies. Over the course of a simulation, the theater can buy supplies, sell products and update the price it sells a particular item. After each update, the theater wants a log of the change to inventory. At the very end of the simulation, the theater would like to know how much money it has. In addition, it would like to have some metrics about the performance of the program.

Since the theater knows you are learning about hash tables in class, they would like for you to implement this functionality via a hash table using separate chaining hashing.

Problem

Write a program that reads in input corresponding to various changes and queries to the theater's inventory and prints out corresponding messages for input commands 1 and 2. Here is an informal list of each of the possible commands in the input:

(1) Buy supplies from a supplier.
(2) Sells a quantity of an item to a customer.
(3) Update the sale price of an item.

For each command, we define the complexity of the command to be as follows:

If the desired item is already in the appropriate linked list, then the complexity of the command is equal to k, where the item is stored in the kth node of the linked list (using 1-based counting).

If the desired item is NOT in the appropriate linked list (meaning that we must create a new node for buying supplies), then the complexity of the command is equal to the length of the linked list plus 1. (This is also the length of the resulting linked list after inserting the new item.)

Your program will compute the sum of complexities of each of the commands, in addition to computing the total amount of cash the theater has after all of the commands are executed.

Input

The first line of input contains a single positive integer: ***n*** (***n*** $\leq$ 300,000), the number of commands to process.

The next ***n*** lines will each contain a single command.

Here is the format of each of the possible commands:

Command 1
`buy <item> <quantity> <totalprice>`

`<item>` will be a lowercase alphabetic string with no more than 19 characters, indicating the item being purchased from the supplier.

`<quantity>` will be a positive integer indicating how many of the item are being purchased.

`<totalprice>` will be a positive integer (in dollars), representing the total purchase price.

Command 2
`sell <item> <quantity>`

`<item>` will be a lowercase alphabetic string with no more than 19 characters, indicating the item being sold to a customer.

`<quantity>` will be a positive integer indicating how many of the item are being sold.

Note: if the stock of the item in question is less than the quantity requested, then just sell all of the available quantity of that item. **It's guaranteed that item will be in the inventory.**

Command 3
`change_price <item> <new_price>`

`<item>` will be a lowercase alphabetic string with no more than 19 characters, indicating a valid item in the inventory.

`<new_price>` will be a positive integer (in dollars), representing the updated price at which the item (single copy) will be sold. **It's guaranteed that item will be in the inventory.**

It is guaranteed that through all of the commands, the theater will never get below $0 and that the total amount of cash the theater has will never exceed the maximum value that can be stored in an integer variable. Same goes for every individual quantity of any item. Also, each item will have a well-defined price before it's sold to a customer. (In short, the data will be such that you can use type int throughout, with minimal error checking except for not selling to much of an item, and all the data will make sense, so to speak.)

Output

For each input command of type 1 and 2, output a single line as described below:

Commands 1 and 2

Print out a single line with the format:

```
<item> <quantity> <totalcash>
```

where `<item>` is the name of the item bought/sold, `<quantity>` is the number of that item left in inventory AFTER the transaction, and `<totalcash>` is the total amount of money left after the transaction.

After all of the transactions have completed, print both the total cash on hand at the end of the simulation on a line by itself, followed by the total complexity of all of the operations executed as previously defined, on a line by itself.

Sample Input	**Sample Output**
10 buy popcorn 1000 3000 buy soda 2000 1000 change_price popcorn 6 change_price soda 5 sell popcorn 50 sell soda 100 change_price popcorn 8 sell popcorn 90 sell soda 1899 buy soda 10 3	popcorn 1000 97000 soda 2000 96000 popcorn 950 96300 soda 1900 96800 popcorn 860 97520 soda 1 107015 soda 11 107012 107012 10

Note: Since popcorn and soda have different hash values, both will be in linked lists of size 1 always and incur a complexity of 1 for each operation. Also, more test cases will be posted before the program is due.

Hash Function to Use

Please use the following hash function:

```
int hashfunc(char* word, int size) {
    int len = strlen(word);
    int res = 0;
    for (int i=0; i<len; i++)
        res = (1151*res + (word[i]-'a'))%size;
    return res;
}
```

Hash Table Insertion Details

When inserting an item into the hash table, insert it to the front of the table. This will affect the cost calculated by your program. (For example, if we insert a new item "popcorn" into the table, and then search for it right afterwards, say a buy followed by a sell, then the cost of the sell will be 1 because popcorn will be at the front of its corresponding linked list.)

Structs to Use

Please use the following #define and two structs in your code:

```
#define MAXLEN 19
#define TABLESIZE 300007

typedef struct item {
    char name[MAXLEN+1];
    int quantity;
    int saleprice;
} item;

typedef struct node {
    item* iPtr;
    struct node* next;
} node;

typedef struct hashtable {
    node** lists;
    int size;
} hashtable;
```

Note: When you initialize your array of node* for your hashtable, please initialize the array dynamically to be the size `TABLESIZE` and set each list in the table to NULL. The size component should simply be set to `TABLESIZE` as well.

Implementation Requirements/Run Time Requirements

1. Use a hash table as previously described, using the hash function previously mentioned.

2. The run-time for processing each of the commands should be amortized O(1) time. The final number printed out will indicate the relative complexity for an input case.

3. A global variable may be used (but doesn't need to be) to keep track of the total complexity and total cash on hand.

4. Your code must compile and execute on the Eustis system. The C compiler on this system is the one that the graders will be using to grade/evaluate your submissions.

Deliverables

1. Please submit a single source file, `inventory.c`, via Webcourses, for your solution to the problem.