Computer Science 1 – Program 2
KnightsMart (Linked Lists)
Assigned: 1/25/12
Due: 2/8/12 (Wednesday) by 11:55pm (WebCourses time)

Objective

Learn to implement the functionality of linked lists.

The Problem

UCF is growing so large that they're actually starting a mini-supermarket, KnightsMart, stocked with typical supermarket products along with all the studying essentials (selection of coffees and creamers, pallets upon pallets of Red Bull, and all the Twinkies you can imagine!).

KnightsMart is in need of a program that:

- manages all products in the store, along with the stock (quantity) for that product
- manages the reordering of depleted products (once the stock for that item reaches zero)
- manages the total sales to all customers in a given day

Throughout any given day, the following things can occur (just as in a normal store):

- New products can be added to the list of products. For example, perhaps the store doesn't initially carry Cheerios. So this new product, Cheerios, can be added to the product line for KnightsMart. Note: when new products are added for the first time, they are initialized with a specific stock amount associated with that product. It is guaranteed that newly added items will always have a quantity of at least one unit.
- Customers can come throughout the day to purchase items from the store. If the items are available, they are purchased, and the stock of those specific products are all reduced by the quantity purchased. If a product is not available (stock is zero), the customer simply does not purchase that product.
- A purchase occurs when a customer finds at least one of the items they are looking for. All such purchases will need to be saved as described later in this write up. However, if a customer enters the store and all of the desired items have a stock of zero, this customer did not make a purchase, and therefore, does not need to be saved.
- As customers purchase products, the quantity for that product is clearly reduced. Once the inventory level for a product reaches zero, that product is immediately added to a restock list.
- Reorders can be made, and they will be based on the restock list. Products that are reordered will have their stock increased based on a fixed restock quantity associated with that particular product. For the sake of ease, whenever a REORDER command is given, we assume the restocking happens immediately (meaning, an order is not placed to some central warehouse, and then we wait for a delivery truck three days later, etc, etc.). So just assume that the items in the restock list are immediately (magically) restocked.

For this program, you will need to maintain three linked lists: a linked list of all products in the store, a linked list of products that need to be reordered, and a linked list of sales that occur on any given day.

You will read in a list of commands from an input file, and your program will execute those commands accordingly. The final command, for each day, will be one that prints out a summary for all sales in the given day, which is detailed in the input/output specifications.

Implementation

You will need to make use of the following structures (**EXACTLY** as shown):

```
typedef struct KnightsMartProduct {
     int itemNum;
     char itemName[21];
     double unitPrice;
     int stockQty;
     int restockQty;
     struct KnightsMartProduct *next;
} KMProduct;

typedef struct KnightsMartRestockProduct {
     int itemNum;
     struct KnightsMartRestockProduct *next;
} KMRestockProduct;

typedef struct KnightsMartSale {
     char firstName[21];
     char lastName[21];
     int numItemsOnList;  // # of items on shopping list
     int *itemsPurchased; // array of item numbers
     struct KnightsMartSale *next;
} KMSale;
```

You will need to make the following three linked lists:

- `KMProducts`: this is an **ordered** linked list of products, where each node is a `KMProduct` (defined above). The products (nodes) of this list are in ascending order based on item number. So the first product in the list is the one whose item number is "smallest", and the last product in the linked list is the one whose item number is largest. As such, insertion into this list must happen at the correct location, thereby maintaining the integrity of the order.
- `KMRestockList`: this is a linked list of the products that need to be restocked. This list is initially NULL. Once the quantity of a product reaches zero, a new `KMRestockProduct` node is made, and it is immediately added to the **end** of this list.
- `KMSales`: this is a linked list of all the sales that occur during a given day, where each node is of type `KMSale`. At the beginning of each day, this list is initially NULL. Once a customer finds and therefore purchases at least one item, that sale is immediately added to the **end** of this list. If a customer enters the store and does not find any of the items they are looking for (all items had a current stock of zero), no sale occurred, and that customer is therefore not added to this list.

Input File Specifications

You will read in input from a file, "**KnightsMart.in**". Have this AUTOMATED. Do not ask the user to enter "KnightsMart.in". You should read in this automatically. (This will expedite the grading process.). The first line of the file will be an integer, *d*, representing the number of days that the simulation will run. The first line of each simulation will be an integer, *k*, indicating the number of commands that will be run for that day's simulation, with each command occurring on a separate line. Each of those *k* commands will be followed by relevant data as described below (and this relevant data will be on the same line as the command).

The commands you will have to implement are as follows:

- ADDITEM – Makes a new product which is added to the store. The command will be followed by the following information all on the same line: `itemNum`, an integer representing the item number for this product; `itemName`, the name of the item, no longer than 20 characters; `unitPrice`, the price of a single item; `stockQty`, the initial amount of stock for this product (guaranteed to be at least 1); `restockQty`, the quantity by which the product should be restocked whenever a reorder is placed. Each item will be unique. Meaning, the input file is guaranteed to not have duplicate items added to the store inventory via the ADDITEM command.

- RESTOCK – This command will have no other information on the line. When this command is read, all items found in the `KMRestockList` are to be restocked immediately. This proceeds as follows:
  - You need to traverse the list using the method shown in class.
  - For every product (node) in this `KMRestockList` list, you need to search for that item in the `KMproducts` list, which will allow you to restock that node accordingly.
  - Once found, you will increase the `stockQty` member of that `KMProduct` based on the `restockQty` member of that same `KMProduct`.
  - You then need to **delete** that particular product (node) from the `KMRestockList`. Note: since you **delete** from the `KMRestockList` as you traverse the list, this means that you will simply be **deleting from the front** of the list at all times.
  - Assuming you follow these steps properly, once you've traversed the entire `KMRestockList`, restocked all necessary products, and then deleted those products from the `KMRestockList`, at this point `KMRestockList` will have no nodes and should point to NULL (meaning, the list is empty).

- CUSTOMER – This command is for customers attempting to make purchases. It is followed by the following on the same line: a first name, then a last name, each no longer than 20 characters; an integer, *n*, representing **two times** (2x) the number of different products the customer wants to purchase, followed by *n* integers, which will be pairs of item numbers and the respective quantity purchased of that item number. For example, **if n were six**, this means the customer wants **three** items. Pretend those following six integers were as follows: 5437 2 8126 1 9828 4. This means the customer wants 2 units of product 5437, 1 unit of 8126, and 4 units of product number 9828.

As mentioned previously, a purchase occurs when a customer finds, and of course then buys, one of the items on their shopping list. When this happens, a `KMSale` node must be made, which records this sale. The struct members must then be filled in accordingly. One member of the `KMSale` node is `itemsPurchased`. This is an array that must be dynamically allocated based on the size (number) on the original shopping list. If the product is found and purchased, the item number is added to the corresponding cell of this array, along with the quantity purchased. We assume that if the stock for a given item is available, the customer will purchase the full desired quantity on their shopping list (of that item). If the product is not found, a zero is recorded in that cell of the array. Based on the three-item example above, if the first and third items were available, with the second being unavailable (no stock), the `itemsPurchased` array would have SIX cells as follows: 5437, 2, 8126, 0, 9828, 4. This shows that 2 units of 5437 were purchased, and 4 units of 9828 were purchased; the zero after item 8126 simply shows that the item was not available. Finally, once all appropriate information is saved into the struct members of the `KMSale` node, this node is then added to the **end** of the `KMSales` list. Note: a line of output is printed regardless of whether or not a purchase was made. Refer to sample output for examples.

- INVENTORY – This command will have no other information on the line. When this command is read, the current KnightsMart inventory (each item with its respective stock) is printed to the file in ascending order (the order that the product list is already in). See sample output file for specific formatting of this command. If there store does not have any inventory, an appropriate error message is printed (see sample output).

- PRINTDAYSUMMARY – This command will have no other information on the line. When this command is read, a report of all sales, for that given day, will be printed to the output file. Please refer to sample output for exact specifications. Once the header is printed (see sample output), all sales (nodes) found in the `KMSales` list will need to be printed, and then those nodes must be deleted. This will proceed as follows:
  - You need to traverse the list using the method shown in class.
  - For every Sale (node) in the `KMSales` list, you need to print Sale number (starting at 1), along with the first and last name of the customer (in that order). You then need to print the list of items that were successfully found and then purchased. Finally, you need to print the total amount paid. Please refer to the EXACT output format for specifics.
  - After printing the appropriate information, you then need to **delete** that particular Sale (node) from the `KMSales` list. Note: since you **delete** from the `KMSales` list as you traverse the list, this means that you will simply be **deleting from the front** of the list at all times. This also means that the first Sale (node) in the list will be both the first Sale printed and the first Sale deleted.
  - Assuming you follow these steps properly, once you've traversed the entire `KMSales` list, printed all necessary information, and then deleted those Sales from the `KMSales` list, at this point the `KMSales` list will have no nodes and should point to NULL (meaning, the list is empty).

- Lastly, you must print out the total sales (dollar amount) for the given day.

Status of Lists at the end of each day:

At the end of each day, the `KMSales` list will clearly be empty, as per the description in the PRINTDAYSUMMARY command. This prints each Sale, deletes each node, effectively destroying that list. The other two lists, `KMProducts` and `KMRestockList`, will be maintained throughout the entire running of your program. Meaning, you clearly should not destroy (delete all nodes from) the list of products in the store at the end of a day, as this would result in there being no products in the store for subsequent days. Additionally, the `KMRestockList` will also be maintained over the simulation. This means that if there were items in the `KMRestockList` at the end of a given day, those items will remain in that list at the beginning of the next day, thereby allowing them to be restocked whenever a REORDER command arrives.

End of Simulation (Memory Cleanup)

All link lists should be destroyed at the end of the simulation before you exit from main.

Output Format

Your program must output to a file, called "**KnightsMart.out**". **You must follow the program specifications exactly.** You will lose points for formatting errors and spelling.

When examining the output file, you should notice that the INVENTORY command and the PRINTDAYSUMMARY command results in printing a “semi-formatted” line of text. To avoid guessing games and to guarantee the correctness of your format, we are providing those two print literals below:

Here is the literal for printing inventory items in the INVENTORY command:

```
"\t| Item %6d | %-20s | $%7.2lf | %4d unit(s) |\n"
```

And here is the literal for printing customer items in the PRINTDAYSUMMARY command:

```
"\t\t| Item %6d | %-20s | $%7.2lf (x%4d) |\n"
```

Sample and Output Files

A sample input file, with corresponding output file, can be found on the course website (the files have too many lines to paste here).

NOTE: These files do NOT test every possible scenario. That is the job of the programmer to come think up the possible cases and test for them accordingly. You can be sure that our graders will grade with very large input files, which are intended to, ideally, test all possible cases. It is recommended that you spend time thinking of various test cases and build your own input file for testing purposes.

*****WARNING*****

Your program MUST adhere to the EXACT format shown in the sample output file (spacing capitalization, use of dollar signs, periods, punctuation, etc). The graders will use very large input files, resulting in very large output files. As such, the graders will use text comparison programs to compare your output to the correct output. If, for example, you have two spaces between in the output when there should be only one space, this will show up as an error even through you may have the program correct. You WILL get points off if this is the case, which is why this is being explained in detail. Minimum deduction will be 10% of the grade, as the graders will be forced to go to text editing of your program in order to give you an accurate grade. Again, your output MUST ADHERE EXACTLY to the sample output.

Grading Details

Your program will be graded upon the following criteria:

1) Adhering to the implementation specifications listed on this writeup.
2) Your algorithmic design.
3) Correctness.
4) **Use of Linked Lists. If your program does not use Linked Lists, you will NOT get credit for the assignment. Period.**
5) The frequency and utility of the comments in the code, as well as the use of white space for easy readability. (We're not kidding here. If your code is poorly commented and spaced and works perfectly, you could earn as low as 80-85% on it.)
6) Compatibility to CodeBlocks. (If your program does not compile in CodeBlocks, you will get a sizable deduction from your grade.)
7) Your output MUST adhere to the EXACT output format shown in the sample output file.

Restrictions

Name the file you create and turn in *KnightsMart.c*. Although you may use other compilers, your program must compile and run using CodeBlocks. Your program should include a header comment with the following information: your name, course number, section number, assignment title, and date. You should also include comments throughout your code, when appropriate.

Deliverables

A single source file named *KnightsMart.c* turned in through WebCourses.