Computer Science I Program #1: Making Smoothies (Dynamic Memory Allocation)

Please check Webcourses for the Due Date

Read all the pages before starting to write your code

A smoothie is a blended drink concoction, with several ingredients. Each different type of smoothie has a different ratio of its ingredients. For example, the "StrawberryBreeze" has the following ratios of items:

2 units strawberry
1 unit banana
1 unit kiwi
2 units yogurt

If a store manager sells 18 pounds of Strawberry Breeze smoothie a week, then his weekly order ought to have 6 lbs of strawberries, 3 lbs of banana, 3 lbs of kiwi and 6 lbs of yogurt.

Of course, there are other types of smoothies a store manager must sell, so to figure out her weekly order, she has to go through how much of each smoothie she'll sell and each smoothie's recipe and put together these calculations.

For your program, you'll read in information about several smoothie recipes and several stores expected sales for the week, and determine what raw ingredients (and how much of each) each store should order.

Note: This assignment test dynamic memory allocation. Later in the write up, requirements will be given stating which memory needs to be dynamically allocated and freed.

The Problem

Given a list of possible smoothie ingredients, a list of smoothie recipes, and lists of sales from several stores, determine how much of each ingredient each store must order.

The Input (to be read from standard input)

The first line will contain a single positive integer, n ($n \leq 10^5$), representing the number of possible smoothie ingredients.

The following n lines will each contain a single string of in between 1 and 20 characters (all letters, digits or underscores). The i^{th} ($0 \leq i \leq n-1$) of these will contain the name of the i^{th} smoothie ingredient. (Thus, the ingredients are numbered from 0 to n-1.)

The next line of input will contain a positive integer, s ($s \leq 10^5$), representing the number of different smoothie recipes. (Use the same numbering convention for the recipes. The recipes are numbered from 0 to s-1.)

The next *s* lines will contain the smoothie recipes, one per line. Each of these lines will be formatted as follows:

$m\ I_1\ R_1\ I_2\ R_2\ \ldots\ I_m\ R_m$

m represents the number of different ingredients in the smoothie ($1 \le m \le 100$)

I_1 represents the ingredient number of the first ingredient ($0 \le I_1 < n$)

R_1 represents the number of parts (ratio) of the first ingredient ($1 \le R_1 \le 1000$) in the smoothie recipe

The rest of the *I* and *R* variables represent the corresponding information for the rest of the smoothie ingredients.

For example, if strawberries were ingredient 7, bananas ingredient 3, kiwis ingredient 6 and yogurt was ingredient 0, then the following line would store a recipe for the Strawberry Breeze previously mentioned:

```
4 7 2 3 1 6 1 0 2
```

The following line of input will contain a single positive integer, *k* ($1 \le k \le 100$), representing the number of stores making orders for smoothie ingredients. (**Number the stores 1 to k, in the order they appear in the input.)**

The last *k* lines of input will contain each store's order, one order per line.

Each of these lines will be formatted as follows:

$r\ S_1\ W_1\ S_2\ W_2\ \ldots\ S_r\ W_r$

r represents the number of different smoothies the store offers ($1 \le r \le s$)

S_1 represents the smoothie number of the first smoothie ($0 \le S_1 < s$)

W_1 represents the weight sold of the first smoothie ($1 \le W_1 \le 1000$), in pounds.

The rest of the *S* and *W* variables represent the corresponding information for the rest of the smoothie ingredients.

Note: the sum of all the number of different smoothies all the stores offer won't exceed 10^6.

The Output (to be printed to standard out)
For each store order, print the following header line:

```
Store #x:
```

where x is the 1-based number of the store.

This will be followed by a list of each ingredient the store must order and the amount of that ingredient (in pounds), rounded to 6 decimal places. The ingredients must be listed in their numeric order (order in the input), one ingredient per line, but instead of printing out the number of the ingredient, print out its name. For example, in the previously listed example, if a store only sold 18 pounds of the Strawberry Breeze and the numbers of the ingredients were 7 - strawberry, 3 - banana, 6 - kiwi, and 0 - yogurt, then the corresponding output (minus the header line) would be:

```
yogurt 6.000000
banana 3.000000
kiwi 3.000000
strawberry 6.000000
```

Thus, the format of each line is:

```
Ingredient_Name Weight_To_Order
```

with weight to order rounded to exactly 6 decimal places.

Sample Input
Since this is lengthy, it will be in the attached file smoothie_sample.in posted on the course webpage.

Sample Output
Since this is lengthy, it will be in the attached file smoothie_sample.out posted on the course webpage.

Implementation Restrictions/ Run-Time/Memory Restrictions
1. The names of each of the ingredients must be stored in a dynamically allocated character array, where the memory for each individual string is ALSO dynamically allocated. (Thus, there should be one malloc.calloc at the top level and several malloc/callocs on the inner level.)

2. You must use the following structs. What these structs store is also described below:

```
typedef struct item {
    int itemID;
    int numParts;
} item;
```

This stores one component of a smoothie recipe. The itemID represents the ingredient number and numParts represents the number of parts of that ingredient.

```
typedef struct recipe {
    int numItems;
    item* itemList;
    int totalParts;
} recipe;
```

This stores one smoothie recipe. numItems stores the number of different ingredients, itemList will be a dynamically allocated array of item, where each slot of the array stores one ingredient from the recipe, and totalParts will equal the sum of the numParts of each ingredient in the smoothie. Notice that this is not an array of pointers but just an array of struct. The reason this design decision has been made is because (a) to give you practice with how deal with an array of struct, and (b) Once the data is read into this array, no changes will be made to the array (ie no swapping of elements), thus, this will work equally well for our purposes as an array of pointers.

3. You must use variables of the following types to perform the following tasks:

(a) All of the smoothies need to be stored an array of pointers to recipes:

```
recipe** smoothieList;
```

Thus, storing the smoothieList will involve one malloc/calloc for an array of pointers. Then, each of those pointers will point to a single recipe.

(b) When processing each store, you must store the amount of each ingredient in a dynamically allocated frequency array. This array should be allocated right before your program reads in the store's sales information (which smoothies it makes and how much of each one). Then, it should be freed right AFTER the calculation of how much of each ingredient needs to be ordered completes and is printed to the screen. This array should look like this:

```
double* amtOfEachItem;
```

(Note: You can figure out how much space to allocate for it. The idea is that amtOfEachItem[i] will store a double equaling the number of pounds for the order for ingredient i.

4. For full credit, you must write the following functions with the following prototypes, pre-conditions and post-conditions:

```
// Pre-condition: 0 < numIngredients <= 100000
// Post-condition: Reads in numIngredients number of strings
//                 from standard input, allocates an array of
//                 strings to store the input, and sizes each
//                 individual string dynamically to be the
//                 proper size (string length plus 1), and
//                 returns a pointer to the array.
char** readIngredients(int numIngredients);
```

```
// Pre-condition: 0 < numItems <= 100
// Post-condition: Reads in numItems number of items
//                 from standard input for a smoothie recipe,
//                 Dynamically allocates space for a single
//                 recipe, dynamically allocates an array of
//                 item of the proper size, updates the
//                 numItems field of the struct, fills the
//                 array of items appropriately based on the
//                 input and returns a pointer to the struct
//                 dynamically allocated.
recipe* readRecipe(int numItems);


// Pre-condition: 0 < numRecipes <= 100000
// Post-condition: Dynamically allocates an array of pointers to
//                 recipes of size numRecipes, reads numRecipes
//                 number of recipes from standard input, creates
//                 structs to store each recipe and has the
//                 pointers point to each struct, in the order
//                 the information was read in. (Should call
//                 readRecipe in a loop.)
recipe** readAllRecipes(int numRecipes);

// Pre-condition: 0 < numSmoothies <= 100000, recipeList is
//    pointing to the list of all smoothie recipes and
//    numIngredients equals the number of total ingredients.
// Post-condition: Reads in information from standard input
//    about numSmoothies number of smoothie orders and dynamically
//    allocates an array of doubles of size numIngredients such
//    that index i stores the # of pounds of ingredient i
//    needed to fulfill all smoothie orders and returns a pointer
//    to the array.
double* calculateOrder(int numSmoothies, recipe** recipeList, int
numIngredients);

// Pre-conditions: ingredientNames store the names of each
//                 ingredient and orderList stores the amount
//                 to order for each ingredient, and both arrays
//                 are of size numIngredients.
// Post-condition: Prints out a list, in ingredient order, of each
//                 ingredient, a space and the amount of that
//                 ingredient to order rounded to 6 decimal
//                 places. One ingredient per line.
void printOrder(char** ingredientNames, double* orderList, int
numIngredients)
```

```
// Pre-conditions: ingredientList is an array of char* of size
//          numIngredients with each char* dynamically allocated.
// Post-condition: all the memory pointed to by ingredientList is
//                 freed.
void freeIngredients(char** ingredientList, int numIngredients);


// Pre-conditions: allRecipes is an array of recipe* of size
//              numRecipes with each recipe* dynamically allocated
//              to point to a single recipe.
// Post-condition: all the memory pointed to by allRecipes is
//                 freed.
void freeRecipes(recipe** allRecipes, int numRecipes);
```

5. You must only declare your string variables **INSIDE** your case loop.

6. The allowable run-time for this program is a bit tricky to state due to the fact that there are many variables. Let n = # of ingredients, k = # of stores ordering, ss = sum of the number of different smoothies in all the orders, and m = max number of ingredients in a smoothie. Then, for full credit, your program must run in $O(n*k + ss*m)$. The program involves one part where you loop through all of the orders and for each order allocate an array of size n, and loop through that array. It also involves, over the course of those k orders, looping through ss total smoothies, each of which could have upto m ingredients.

Deliverables

You must submit four files over WebCourses:

1) A source file, *smoothie.c*. Please use stdin, stdout. There will be an automatic 10% deduction if you read input from a file or write output to a file.

2) A file describing your testing strategy, *lastname_Testing.doc(x) or lastname_Testing.pdf*. This document discusses your strategy to create test cases to ensure that your program is working correctly. If you used code to create your test cases, just describe at a high level, what your code does, no need to include it.

3) Files *smoothie.in* and *smoothie.out*, storing both the test cases you created AND the corresponding answers, respectively. **(Note: 3 or 4 of your own test cases are expected and all of them can be hand made, so it's not necessary to make a max case to get full credit here.)**