CSCI-1200 Data Structures — Spring 2022
Homework 4 — Pizza Event SimulaTOr

In this assignment you will write a program to manage the kitchen and orders for an imaginary (and very poorly planned) pizza store. Your program will handle several different operations: adding new orders (which will 'expire' if not filled fast enough), adding new items to what's being cooked in the kitchen, managing completely cooked items, various output functions, and advancing either until the next "event" (see description below) or until a certain amount of time has passed in the simulation. Note that when we say "time" we are talking about an artificial measurement of minutes in your simulator. It has nothing to do with real world time or how long your program runs for.

We provide the code to parse an input file that specifies these operations. Furthermore, we provide an implementation of the code that checks if the kitchen has cooked all the items that an order requires. You will use the STL `list` class heavily in your implementation of this program. *Please carefully read the entire assignment before beginning your implementation.*

The most important operation is the *run_until_next* operation which is one of two commands, along with *run_for_time* that actually causes events (kitchen finishes cooking an item, an order runs out of time, an order is filled) to be processed. All other commands just add new orders to the program or add new items to the kitchen.

The kitchen is unaware of what is being ordered by customers, and cooks whatever the *add_item* commands tell it to. Items from the kitchen are not associated with a specific order until the entire order can be filled. This means that if you order two `small_chicken_pizza`s and someone else after you only orders one `small_chicken_pizza`, then if the kitchen only makes one `small_chicken_pizza` the second smaller order – not yours – would take the pizza, even though you ordered first. This is a poor design that can lead to large orders being unfilled for a long time. (And a design that gets used in some major pharmacies!)

A formal explanation of the priority for events is in the *run_until_next* description below.

The input for the program will be read from `cin` and written to `cout`. Here's an example of how your program will be called:

```
./pesto.out < input.txt > output.txt
```

The form of the input is relatively straightforward. Each request begins with a keyword token. There are eight different requests, described below. You may assume that the input file strictly follows this format (i.e., you don't need to worry about error checking the formatting).

On the next page there are details on each command.

run_until_next will advance time until the next event occurs, and resolves the event. This includes updating the remaining time on all orders and cook time on items being cooked. If there are no events, "No events waiting to process." should be printed. Otherwise, the process for finding an event is as follows, with the highest priority listed first (used for choosing which event if multiple events are tied for "soonest"):

1. If a kitchen item that is being cooked can be completed, it should be removed from the list of items being cooked and added to the end of the list of items that are completely cooked.

2. If an order can be fulfilled (starting with lowest promised time remaining), that order is resolved. For this to happen, all items (be careful – there might be duplicates) required by the order must be in the list of completely cooked items. If an order is resolved, it is removed from the orders list, and all items that were used (prefer oldest first if there's duplicates) are removed from the completely cooked item list.

3. If an order cannot be fulfilled and it has 0 minutes remaining, then it expires and should be removed from the orders list.

run_for_time ***run_time*** advances the time by *run_time* minutes, resolving all events (as described above in the `run_until_next` description) that could happen in that timespan. It also updates the times remaining on orders and items that are cooking.

add_order ***order_ID promised_time number_of_items item1 item2 ... itemN*** is used to add a new order to the list of pending orders. This list should be kept sorted by shortest time remaining, with ties resolved by order ID. The ID is a positive number and you can assume our input does not contain any duplicate IDs. The promised time is the number of minutes the store has promised to complete the order in, this will always be 0 or larger. The rest of the command is a count of items in the order, followed by strings describing each item.

add_item ***cook_time item_name*** adds a new item to the kitchen's list of items being cooked. *cook_time* specifies how long is left before an item is considered completely cooked and will always be 0 or larger. If cook_time is 0, the `add_item` command still should add the item to the "being cooked" list. This list is sorted by smallest remaining cook time, with ties being broken by which name comes first alphabetically.

print_orders_by_time outputs the order list sorted as described in the `add_order` command.

print_orders_by_id outputs the order list sorted by smallest ID first. Since IDs are unique, there is no behavior specified for ties.

print_kitchen_is_cooking outputs the list of items that have not finished cooking (can include items with 0 minutes left), in the order described in the `add_item` command.

print_kitchen_has_completed outputs the items that have been completely cooked in the order they were completed.

All of the normal output of the program should be written to `cout`. Please consult the provided output files carefully as they demonstrate output formatting that may not be described in this handout.

Assignment Requirements, Hints, and Suggestions

- **You may not use vectors or arrays for this assignment.** Use STL lists instead. You may not use maps, or sets, or things we haven't seen in class yet.
- We have provided a partial implementation of this program, focusing on input parsing and an algorithm for determining if an order can be fulfilled. There are member function calls to our versions of the `Order` and `Item` classes, so you can deduce how some of the member functions in our solution work. You are *strongly encouraged* to examine this code carefully and follow the interfaces suggested in the provided code.
- In your README.txt file, provide an order notation analysis of each operation. Refer to the provided README.txt for which variables your order notation can use.
- Do all of your work in a new folder named `hw4` inside of your homeworks directory. Use good coding style when you design and implement your program. Be sure to make up new test cases to fully test your program and don't forget to comment your code! Use the template `README.txt` to list your collaborators, your time spent on the assignment, and any notes you want the grader to read. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your README.txt file.
- To encourage your fluency with the traditional step-by-step debugger we will post an additional Crash Course in C++ module: "Lesson 12: Debugger Use". Completing this module before Friday February 18th at 11:59pm will be worth a *small* number of extra credit points on Homework 4.