# Consolidated Offers Programming Exercise

## Introduction

**Disclaimer**: You do **not** need to implement anything that is described in the Introduction!

In a fictional region called Galaland[1], a large number of apple farmers are offering their apples for sale at their farms. Recently, the apple industry was modernized, and farmers now use an electronic system. The system keeps track of each individual farmer's offers, and updates it when a customer makes a purchase.

For the purpose of this exercise, *offers* consist of

- a quantity of apples,
- a minimum selling price in Gala dollars ($), and
- an offer identifier,
  Which comprises an integer refNum followed by a farmer's identifier: e.g. "1:A", "17:A", "18:B". The refNum is unique across all farmers, and increases by 1 for each new offer.

The offers are ranked: an offer that is cheaper is better than a more expensive one, and for equal prices, a lower refNum is preferred (as the offer is older). All offers of a particular farmer are collectively stored in such a ranked *offer list*.

| Offers ↓ |
|---|
| 10 apples at $21 ref 12:A |
| 1 apple at $19 ref 13:A |
| 2 apples at $19 ref 11:A |
| 3 apples at $18 ref 10:A |
| 1 apples at $17 ref 3:A |

Figure 1 - Example of an offer list

| Offers ↓ |
|---|
| 10 apples at $21 ref 12:A |
| 1 apple at $19 ref 13:A |
| 1 apple at $19 ref 11:A |

Figure 2 - Offer list after a customer purchase of 5 apples for $20

| Offers ↓ |
|---|
| 10 apples at $21 ref 12:A |

Figure 3 - Offer list after a second customer attempts to purchase 4 apples for $19, but only gets 2

For instance, if a customer wanted to purchase 5 apples for (no more than) $20, the offers 3:A and 10:A would be *fully executed*, meaning all of their apples were sold and they disappeared off the list. 11:A would be *partially executed*, meaning some apples were not sold, and it remains on the list as shown in Figure 2. In Figure 3, another customer wanted to buy 4 apples at an even better price, $19, but farmer A agreed to sell only two apples at that price, so the purchase was partially successful.

[1] Inspired by the Gala apple cultivar, which is the cultivar with the highest production in the United States.

## Iceberg Offers

Sometimes, farmers prefer if other farmers can't see how many apples they offer. For this purpose, a special offer type was introduced, called *iceberg* offer. Iceberg offers don't disappear off the list when they are fully executed. Instead, they are restocked (i.e. their apple quantity is refilled) and reinserted into the list. They have a visible number of apples, and a hidden total number of apples.

In the code, we represent

- the refill size by (`const`) `Offer::displayedQtyMax`,
- the total number of apples by `Offer::qty`, and
- the apples currently displayed on the offer list by `Offer::displayedQty`, which is replenished to `min(displayedQtyMax, qty)` upon being emptied.

Normal offers simply have `displayedQtyMax` as infinity, so all `qty` apples are displayed: `Offer::qty == Offer::displayedQty`.

For instance, Figure 4 shows an offer list with an iceberg offer 10:B, with a refill size of 3 and a total size of 50 apples. Now, a customer purchase for 10 apples at $20 comes in:

| Offers ↓ |
| --- |
| 10 apples at $21 ref 12:B |
| 2 apples at $19 ref 11:B |
| 3(/50) apples at $19 ref 10:B |

Figure 4 - Offer list

| Offers ↓ |
| --- |
| 10 apples at $21 ref 12:B |
| 3(/47) apples at $19 ref 13:B |
| 2 apples at $19 ref 11:B |

Figure 5 - Intermediate offer list state after one execution

| Offers ↓ |
| --- |
| 10 apples at $21 ref 12:B |
| 1(/42) apples at $19 ref 15:B |

Figure 6 - Final offer list state

In Figure 5, 10:B's visible tip was restocked to 3 apples, and reinserted as 13:B (a new offer ID) behind 11:B. Next, 11:B is fully executed. The remaining 5 apples are sold to the customer by 13:B in two executions, which thus remains in the offer list as 15:B. The final list is shown in Figure 6.

Just like regular offers, farmers can have multiple iceberg offers at the same price.

## The Task

The goal of this exercise is to maintain a *consolidated offer list*, which combines all the individual farmers' offer lists into a single list, so that customers can see the collective state of the apple market in Galaland. This is what it might display at a given point in time:

| Offers ↓ |
|---|
| ... |
| 10 apples at $21 ref 12:B |
| 7 apples at $20 ref 11:D |
| 10 apples at $20 ref 10:D |
| 2 apples at $19 ref 16:B |
| 3 apples at $18 ref 2:B |

Figure 7 - A consolidated offer list including the offers for farmer B from Figure 5, among some unrelated offers of farmers C and D

The consolidated list ranks the offers just like a regular list does; by price first, then by refNum, both decreasing. I.e. offers with lower prices appear after offers with higher prices, and within the same price level, offers with lower refNums appear after offers with higher refNums. The best offers reside at the end of the list.

**When a customer makes a purchase, they never purchase from the consolidated list; instead, they purchase from a specific farmer.** That means that the consolidated list changes not only at the end, where the overall best offers are kept, but anywhere in the middle as well.

When a new offer is entered, it appears in the consolidated list before all other offers at the same or a better price. Its refNum will be the highest in the list.
When an iceberg offer is restocked, it's reinserted like a new offer, with a new refNum, which also means that it reappears before any offer at the same (or better) price.

The users of this system have the peculiar requirement of both fast serialization of the consolidated list, as well as fast updates on it (as new purchases are made). The data structure chosen for the list takes care of the former, and your implementation of the updates should adhere to the latter!

## Implementation Details and Constraints

The consolidated list has type `vector<PublicOffer>` (named `ConsolList`), with the best offers at the end, i.e. the `vector` is sorted, and the order is from worst to best offer (lowest prices at the end of the vector). This data structure is fixed, and cannot be changed. `PublicOffer` is almost identical to `Offer` and can be constructed from it. It only stores the displayed size, not the total one.

The consolidated offer list is maintained during a purchase via the two classes `ConsolListEntryUpdater` and `ConsolListPurchaseUpdater`. There are five member functions that must be implemented:

- `static ConsolListEntryUpdater::insert(ConsolList& cl, const Offer& o):`
  called to update `cl` with a new offer `o`.
  This function's implementation is unrelated to the below functions' implementations.

- `ConsolListPurchaseUpdater::onPurchaseBegin()`:
  Called when a purchase process is started.
- `ConsolListPurchaseUpdater::onMatch(const Offer& o)`:
  Called successively when an offer is executed. Any matched apples are already subtracted from the member variables `qty` and `displayedQty` of `o` at the point of the call, but icebergs will not yet have been restocked.
- `ConsolListPurchaseUpdater::onReentry(const Offer& iceberg)`:
  called when an iceberg offer has matched all of its visible apples, but has hidden ones left, and is restocked and reinserted into the offer list. Its member variables have already been updated to reflect the new state.
- `ConsolListPurchaseUpdater::onPurchaseEnd()`:
  Called once the purchase process is finished. After this function is called, the `ConsolList` must be in a good state, but it can be dirty before that.

Each time an offer is entered, `ConsolListEntryUpdater::insert` is invoked.
Then, when a purchase is performed, `ConsolListPurchaseUpdater` is constructed, `onPurchaseBegin()` called, and **its member functions are repeatedly called in the exact order in which the individual offers are matched, best to worst**:

- `onMatch` will always be invoked on the farmer's best offer in their own list (if it matches), and
- when the last call to `onMatch()` had `displayQty == 0` but also `qty > 0` (i.e. this is an iceberg), it is followed by `onReentry` for that same offer.

When the purchase is finished, `onPurchaseEnd()` will be called, during which any dirty state in the `ConsolList` can be cleaned up. The `ConsolList` should be in a valid state after `onPurchaseEnd()` has been called, after which the updater is destroyed. `ConsolListPurchaseUpdater` is expected to keep member variables in order to efficiently and correctly perform any modifications on the `ConsolList`.
Assume that the `ConsolList` can be very large, encompassing many thousands of farmers whose offers are scattered across the `ConsolList`. **You will have to think carefully about the necessary modifications and iterations within the `ConsolList`**, as erasure and insertion cost in a `vector` scales with size.

You are furthermore provided with a print function for `ConsolList`, and a convenience function `verify(..)`, which can be used to ascertain whether a farmer's offer list is correctly represented within a `ConsolList`. See `Test.cc` for a sample usage.

You should not modify the interface of the five declared member functions, the data structure of `ConsolList`, and in general any files except `ConsolListUpdaters.h/.cc` and `Test.cc` (but you are free to add new files/functions if needed).

## Evaluation

Your solution will be evaluated for correctness, efficiency and readability, roughly in that order. Regarding the efficiency criterion, the solution will be evaluated for two scenarios:

- simple purchases, which either don't change the structure of the list at all, or only cause reentries, and
- large purchases, which execute (i.e. erase) many offers in a list at once.

You should be able to explain your algorithm and defend its runtime.

You are also expected to test your solution. We encourage you to test with your own test cases, although some expected behavior will be provided in the `Test.cc` file for convenience.