

CS1020E: DATA STRUCTURES AND ALGORITHMS I

Lab 3 – Distributor

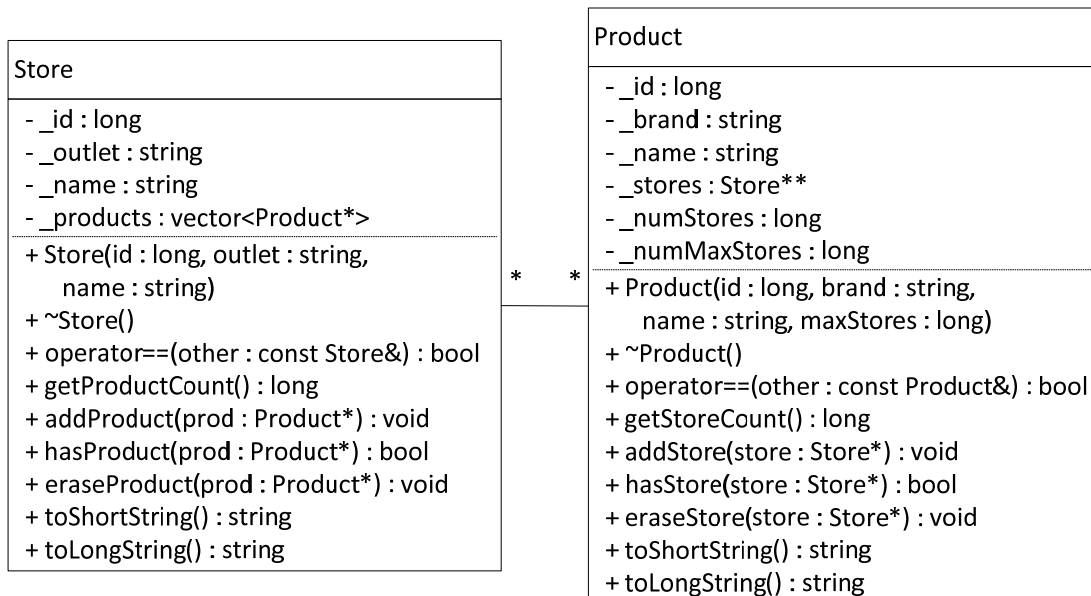
(Week 5, starting 5 September 2016)

Readme

You are given the design and header files for a system. This system is supposed to help a distributor keep track of the stores they deal with, the products the distributor has, and which stores each product is sold at. Complete the implementation of the **Store** and **Product** classes, and use them to achieve 2 to 3 sub-tasks:

1. Handle stores and product with *limited* functionality 65%
2. Handle stores and products with *full* functionality 30%
3. *Additional* functionality - Efficiently find the number of distribution groups 5%

You have to submit the three subtasks separately, but each successive subtask will add to the functionality of the preceding subtasks.



For simplicity, brands, outlets, and names, are each **one-worded**, and only consists of letters. The id of a store/product is a **positive** long integer. Each of the following is **unique within itself**:

- store id
- (store outlet, store name) pair
- product id
- (product brand, product name) pair

i.e. there may be a store having the same id as a product, but no two stores have the same id; also, two stores may have the same outlet but different store name, or have the same name but are different outlets.

The distributor does not want to sell each product in too many different outlets. There will be a system-wide **cap** imposed on the **maximum number of stores** in which **each product** can be sold.

The system should support the following 7 functionalities:

- add a **new store** to system `+store <id> <outlet> <storeName>`
- add a **new product** to system `+prod <id> <brand> <productName>`
- **remove a store** from system `-store <id>` OR `-store <outlet> <storeName>`
- **remove a product** `-prod <id>` OR `-prod <brand> <productName>`
- **map** product to store `+map <brand> <productName> <outlet> <storeName>`
- **un-map** product from store `-map <brand> <productName> <outlet> <storeName>`
- **print all** products and stores after all input has been read and processed

For example, if we want to distribute 'KitKat Chocolate' to 'Hougang NTUC', then we have to:

1. add a store `+store 555 Hougang NTUC`
2. add a product `+prod 1337 KitKat Chocolate`
and then
3. map product to store `+map KitKat Chocolate Hougang NTUC`

Input/Output Format

The first line in the input contains $2 \leq M \leq 3,000$, the maximum number of stores a product can hold. Each subsequent line contains one of the 6 types of operations explained above.

Each operation will output its result:

- when adding a **new store** to the system `Store <short description> added`
- when adding a **new product** to the system `Product <short description> added`
- when **removing a store** to the system `Store <short description> removed`
- when **removing a product** to the system `Product <short description> removed`
- when **mapping** a product to a store `Product <prod short desc> mapped to store <store short description>`
- when **un-mapping** a product from a store `Product <prod short description> unmapped from store <store short description>`

A store/product's **short description** is:

`-<id> <outlet/brand> <name>-`

A store's (similar for product) **long description** is:

`=Store <short description>`

`=<p> product(s) [`

`= <prod short description> |`

Repeat this line p times

`=] 4 spaces`

The products should be stored and output in the order they were mapped to the store.

If $p == 0$, i.e. the store has no products, just display 0 products without square brackets (`[]`)

`=0 product(s)`

When **printing all** products and stores at the end of all operations, the system will output:

blank line

<i> store(s) remaining

blank line

<store long description>

Repeat these two lines i times

blank line

<j> product(s) remaining

blank line

<prod long description>

Repeat these two lines j times

The stores and products should be stored and output in the order they were added. Within a store/product, the mappings should be displayed in the order they were mapped.

Start by viewing dist1.in and dist1.out for sample input and output.

Submission

Submit source files distributor.cpp, product.cpp and store.cpp **ONLY**. The 2 **header files** are **already in CodeCrunch**, and will NOT be uploaded. If you are attempting **x** parts, ensure you submit to the first **x** tasks!

Limited Functionality - Manage Stores & Products



65%

Just for this part, you can be assured that:

- when **adding** a new store/product to the system, the given store/product
 - will not already exist
- when **removing** a store/product from the system, the given store/product information
 - will match an existing store/product
 - will not have mappings to any product/store
- when **mapping** a product to a store
 - the given product will exist
 - the given store will exist
 - the number of stores the given product is sold at has not yet been maxed out
 - the product has not already been mapped to the store
- when **unmapping** a product from a store
 - the given product will exist
 - the given store will exist
 - the mapping between the given product and store will exist

Tip

Don't start coding right away!

Understand the problem: Think how memory would look like when there are a few stores, products, and mappings in the system. Then, think of what each member function of Product and Store *should* do, but you don't implement it yet. Next, identify what your program should do in each operation.

Design: Come up with an algorithm for each operation, tracing what really happens in memory at each step. Verify that your understanding is correct, that your algorithm for each operation does its job correctly and completely. Finally, develop the algorithm for each member function of Product and Store.

Incremental coding and testing: Do not code the entire program at once. Implement Product and Store member functions first, and test them in main(). Once you are satisfied, develop the printAll() member function of your system, so that you are able to check whether the output at the end is somewhat correct. Then, code the other member functions one by one and test incrementally.

Full Functionality - Defensive Programming



30%

People often make mistakes when entering input into a program. For this part, you can still assume that the input format is valid, and that each field is one-worded. However:

- when **adding** a new store/product to the system, the given store/product
 - may already exist in the system - same id, or same (outlet/brand, name) pair
- when **removing** a store/product from the system
 - the given store/product information may not match an existing store/product
 - the matched store/product may have mappings to a product/store
- when **mapping** a product to a store
 - the given store/product information may not match an existing store/product
 - the number of stores the given product is sold at may have been maxed out (at capacity)
 - the product is already mapped to the store
- when **unmapping** a product from a store
 - the given store/product information may not match an existing store/product
 - the mapping between the given product and store may not exist

When **adding** a new store/product, there may be up to two existing matches. Only provide information about the **first match**, i.e. the matching store/product that was added earlier.

When **mapping** a product to a store, you should only check for existing mapping when the number of stores the given product is sold at has not yet been maxed out, i.e. **follow the order given above**.

Output Formats

Adding store/product - Already exists in system

Store <1st match's short> already exists, cannot add <given short>

Product <1st match's short> already exists, cannot add <given short>

Removing store/product - Match does not exist in system

No such store

No such product

Removing store/product - Has mappings to product/store

Cannot remove, store <short description> is mapped to product(s)

Cannot remove, product <short description> is mapped to store(s)

Mapping product to store - Matching product or store does not exist in system

Cannot map, no such product or store

Mapping product to store - Product has number of stores maxed out

Product <product short> maxed out, cannot map to store <store short>

Mapping product to store - Product already mapped to store

Product <product short desc> already mapped to store <store short>

Un-mapping product to store - Matching product or store does not exist in system

Cannot unmap, no such product or store

Un-mapping product to store - Product not mapped to store

No existing mapping between product <product short description> and store <store short description>

Sample Input/Output

See `distri[2..3].in` for sample input and `distri[2..3].out` for the corresponding expected output

Efficiently Find Distribution Groups

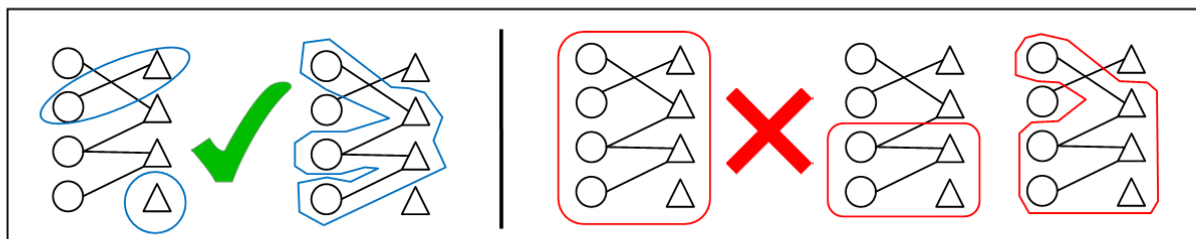


You now use the same system for another distributor who has many $0 \leq \text{stores} \leq 3,000$ and

$0 \leq \text{products} \leq 3,000$. In addition to the full functionality, you are to help the distributor to be able to query how many distribution groups are needed.

A **distribution group** is a set of products and stores such that:

- every store outside of this distribution group is NOT mapped to any product in this group
- every product outside of this distribution group is NOT mapped to any store in this group
- there are no smaller distribution groups within a distribution group



For **N** products and **N** stores, there may be up to N^2 mappings between products and stores. This algorithm is too inefficient as proportional to N^3 products/stores are examined ($O(N^3)$ or cubic time):

```
assign distribution group id 1..N, one for each store
assign distribution group id N+1..2N, one for each product
for each mapping
    oldGrp = max(store group id, product group id)
    newGrp = min(store group id, product group id)
    if oldGrp == newGrp then continue examining next mapping
for each store
    if store group id == oldGrp
        store group id = newGrp
for each product
    if prod group id == oldGrp
        prod group id = newGrp
count the number of unique group ids
```

Header File Changes

We just need up to two additional containers to do the job. Therefore, just for this part, the Store class defined in store.h and Product class defined in product.h now have an additional member variable and 3 member functions each.

```
class Store {
    long _group; // distribution group id
    ...
public: ... // add all of current store's products to the vector
    void appendAllMappedProds(vector<Product*>& someProducts);
    void setGroup(long group);
    long getGroup();
};

class Product {
    long _group; // distribution group id
    ...
public: ... // add all of current products's stores to the vector
    void appendAllMappedStores(vector<Store*>& someStores);
    void setGroup(long group);
    long getGroup();
};
```

Input/Output Format

If the query is ?groups, output a line:

<#> distribution group(s)

If the query appears once or more in the program, **do NOT** execute the **print all** operation.

- End of Lab 3 -