Práctica PRO2 Primavera 24-25

Comerciar

Para crear la función **comerciar** utilice una estructura iterativa dentro de la clase ciudad a través de dos iteradores “it” y “it2” que recorrerían los inventarios del objeto implícito y el por referencia.

Inicio los iteradores al principio de cada Inventario de cada ciudad. Esto lo hago para poder recorrerlos de forma eficiente y teniendo como máximo de iteraciones el número de productos que hay en la ciudad 1 mas los de la ciudad 2.

La condición de salida para el bucle es que un iterador llegue al final de su inventario. Ya que cuando esto suceda las ciudades ya habrán comerciado todos los productos posibles y no podrán comerciar más.

Dentro del bucle lo primero que hago es ver si el producto que señalan los iteradores son el mismo: if (it->first == it2->first)

Si no son el mismo, no se ejecuta el comercio del producto y pasa a la zona donde se comprueba cuál de los dos iteradores debe incrementarse según cual de estos dos tiene el id menor:

```
        else if(it->first < it2->first) ++it;
        else ++it2;
```

Si los iteradores son el mismo se incrementan ambos en uno.

```
            ++it;
            ++it2;
```

Utilizo la función consultar_sobrante para que me obtener la resta de los en posesión menos los necesitados del producto para cada ciudad y los almaceno en los int n1 y n2.

Luego de eso procedo a calcular los posibles escenarios con los que se puede encontrar:

- Que alguna de las ciudades tenga justo los que necesita y no quiera comerciar: En este caso la iteración no generara cambios en los inventarios.

- Que una ciudad le falte más que los que le sobran a la otra: En este caso la primera ciudad se quedara con todos los que le sobren a la otra.
- Que a una ciudad le sobren más que los que le falten a la otra: En este caso la ciudad le dará exactamente el número que le falte a la otra.

En el código se observa que se tratan los 3 posibles casos de forma simétrica para las dos ciudades y para todos los objetos que pertenezcan a las dos ciudades.

Para modificar cada producto comerciado se utiliza la función: **`actualizar_producto()`**

Esta función simplemente augmenta o decrementa las unidades del producto según la cantidad que se le pone y también actualiza su peso y volumen total de forma acorde.

Así, la función comerciar se ejecuta correctamente y de forma iterativamente eficiente.

El código del programa es el siguiente:

```
void Ciudad::comerciar_ciudad(Ciudad& cd, const Total_productos& tp){

    map<int,pair<int,int>>::iterator it = InvCiudad.begin();
    map<int,pair<int,int>>::iterator it2 = cd.InvCiudad.begin();

    while (it!=InvCiudad.end() and it2!=cd.InvCiudad.end())
    {
        if (it->first == it2->first)
        {
            int c1 = consultar_sobrante(it->first);
            int c2 = cd.consultar_sobrante(it->first);
            if (c1<0 and c2>0)
            {
                if(c1*(-1) >= c2){
                    actualizar_producto(it->first,c2,tp);
                    cd.actualizar_producto(it->first,-c2,tp);
                }
                else if (c1*(-1) < c2)
                {
                    actualizar_producto(it->first,-c1,tp);
                    cd.actualizar_producto(it->first,c1,tp);
                }

            }
            else if (c1>0 and c2<0)
            {
                if(c1 >= c2*(-1)){
                    actualizar_producto(it->first,c2,tp);
                    cd.actualizar_producto(it->first,-c2,tp);
                }
                else if (c1 < c2*(-1))
                {
                    actualizar_producto(it->first,-c1,tp);
                    cd.actualizar_producto(it->first,c1,tp);
                }
            }
            ++it;
            ++it2;
        }
        else if(it->first < it2->first) ++it;
        else ++it2;
    }
}
```

Ruta de hacer viaje

En la función de hacer viaje uso una subfunción llamada **rec_viaje** que tiene como objetivo modificar el vector por referencia que se le pasa y ponerle las ciudades por las que este pasa ordenadas de manera cronológica.

La especificación de la función es:

```
int rec_viaje(const BinTree<string>& bt, Barco& bc, int vendidas, int
compradas, vector<string>& v) const
```

Para cada llamada recursiva la función calculará lo provechosa que es una ruta, luego buscará lo hará con su hijo izquierdo y derecho y finalmente las irá comparando hasta sacar la ruta deseada.

La operación tiene dos casos en los que detendrá su recursividad:

1. Si el árbol tratado es vacío.
   En este caso se devuelve la suma de lo vendido y comprado hasta el momento.

```
    if (bt.empty())
    {
        return vendidas+compradas;
    }
```

2. Si después de ver la interacción de la ciudad con el barco el número de compradas y vendidas es máximo.
   En este caso se añade la ciudad al vector y se devuelve la suma de vendidas y compradas.

```
    if (vendidas == bc.max_vender() and compradas == bc.max_comprar())
    {
        v.push_back(bt.value());
        return vendidas+compradas;
    }
```

Después de calcular la posible interacción de cada nodo con una ciudad, de manera similar a como lo hacíamos en el interior del bucle de comerciar, lo que se hace es crear dos vectores; vl y vr. Que se usaran para llamar de manera recursiva a la función para las ciudades hijas izquierda y derecha respectivamente.

```
      vector<string> vl;
```

```
    vector<string> vr;
  int a = rec_viaje( bt.left(),  bc, vendidas, compradas, vl);
  int b = rec_viaje( bt.right(), bc, vendidas, compradas,vr);
```

Finalmente hace las comparaciones de las ciudades hijas para designar cuál de ellas será la que se elija a través de los criterios designados. La ruta elegida se añade a la padre y se devuelve el valor de las vendidas más las compradas.

Se elige la ruta a través de condicionales "if" que comprueban respectivamente los siguientes casos:

- Primeramente, se mira si existen rutas hijas o si estas son vacías. Si las dos son vacías no se modifica la ruta y se devuelve el valor de las vendidas más las compradas.
- Luego se comprueba si existe una ruta más provechosa que la otra. Si existe se escoge esta.
- Si las dos rutas son igual de provechosas se elige la que es más corta.
- Si las dos son iguales, se elige la del hijo izquierdo.

Dentro de estas comparaciones se mira si estas forman o no parte de la mejor ruta o, por el contrario, no se ha comerciado nada dentro de esta. Para saber si añadir el camino miramos si ha generado alguna interacción o si sus recorridos hijos lo han hecho. Si la ruta contiene alguna ciudad, no es necesario poner un if especifico, de lo contrario se hace a través de este "if":

```
if(vendido or comprado) v.insert(v.begin(),bt.value());
```

Por inducción se ve que:

h → altura del árbol

nodo → nodo del árbol

Hipótesis → ∀ nodo ⇒ ∃ vl ∧ ∃ vl

Para el caso vacío (h=0):

(bt.empty()) ⟹ (∄ nodo)

Para la tesis (h!= 0)

(not bt.emty()) ⟹ (∃ vl ∧ ∃ vr)

Viendo la tesis y el caso inicial se deduce por inducción que la función decrementa y lo hace en función de la altura del árbol.

La implementación de la función es la siguiente:

```
int Mapa::rec_viaje(const BinTree<string>& bt, Barco& bc, int vendidas, int
compradas, vector<string>& v) const{
    if (bt.empty())
    {
        return vendidas+compradas;
    }
    bool vendido = false;
    bool comprado = false;

    if (vendidas < bc.max_vender())
    {
        if (mapa.at(bt.value()).consultar_sobrante(bc.ret_id_v())<0)
        {

            if(mapa.at(bt.value()).consultar_sobrante(bc.ret_id_v())*(-1) >=
bc.max_vender()-vendidas){
                vendidas = bc.max_vender();
                vendido = true;
            }
            else if (mapa.at(bt.value()).consultar_sobrante(bc.ret_id_v())*(-1) <
bc.max_vender()-vendidas)
            {
                vendidas +=
mapa.at(bt.value()).consultar_sobrante(bc.ret_id_v())*(-1);
                vendido = true;
            }
        }
    }
    if (compradas < bc.max_comprar())
    {
        if (mapa.at(bt.value()).consultar_sobrante(bc.ret_id_c())>0)
        {

             if(mapa.at(bt.value()).consultar_sobrante(bc.ret_id_c()) >=
bc.max_comprar()-compradas){
                compradas = bc.max_comprar();
                comprado = true;
            }
            else if
(mapa.at(bt.value()).consultar_sobrante(bc.ret_id_c())<bc.max_comprar()-compradas)
            {
                compradas +=
mapa.at(bt.value()).consultar_sobrante(bc.ret_id_c());
                comprado = true;
            }
        }
    }
    if (vendidas == bc.max_vender() and compradas == bc.max_comprar())
```

```
    {
        v.push_back(bt.value());
        return vendidas+compradas;
    }

    vector<string> vl;
    vector<string> vr;
    int a = rec_viaje( bt.left(),  bc, vendidas, compradas, vl);
    int b = rec_viaje( bt.right(), bc, vendidas, compradas,vr);

    if (vl.size()!=0 or vr.size()!=0) {
        if (a > b)
        {
            v=vl;
            v.insert(v.begin(),bt.value());
            return a;
        }
        else if (a < b)
        {
            v=vr;
            v.insert(v.begin(),bt.value());
            return b;
        }
        else
        {
            if (vl.size()>vr.size())
            {
                v=vr;
                v.insert(v.begin(),bt.value());
                return b;
            }
            else
            {
                v=vl;
                v.insert(v.begin(),bt.value());
                return a;
            }
        }
    }
    if(vendido or comprado) v.insert(v.begin(),bt.value());
    return vendidas+compradas;
}
```