Justificación de operaciones PRO2

Mounaim Chakroun GRUPO: 41

25 de mayo de 2024

1 Justificación de Comercio

1.1 Especificación de Comerciar:

```
/**
* @brief Realiza comercio con otra ciudad.
*
* @param Productos Conjunto de productos disponibles.
* @param ciudad_con_la_que_comerciaremos Referencia a la
    ciudad con la que se comerciar .
*
* @pre El inventario de la ciudad puede estar vac o o
   no y debe haber otra ciudad con la que comerciar.
* @post Se realiza el comercio entre las dos ciudades.
*/
void comerciar_c(Cjt_Productos& Productos, Ciudad&
   ciudad_con_la_que_comerciaremos);
```

Listing 1: comerciar_c (Ciudad)

1.2 Paso 1 (inicialización):

Cuando entramos en la función **comerciar_c**, el puntero `_it_Inventario` (propio de la función) se inicializa en la posición inicial del `_Inventario` (inventario de la ciudad), es decir, apuntamos al primer elemento del `map` (si existe).

Entonces, al comprobar la condición del `while` principal, si el iterador `it_Inventario` apunta al `end` del inventario `_Inventario` de `this` entonces terminamos, en caso contrario procedemos a hacer los cálculos.

1.3 Paso 2 (mantenimiento):

1. Primero miramos si nuestro producto seleccionado (el producto apuntado por nuestro `_it_Inventario`) existe en el inventario de la

ciudad con la que queremos comerciar. Para esta tarea llamamos al iterador propio de la ciudad con la que comerciamos y le pedimos que busque el producto deseado.

En este punto puede suceder uno de los dos casos siguientes:

- El producto no se encuentra en el inventario de la ciudad con la que comericamos:

  Entonces pasamos al siguiente elemento acualizando el iterador `_it_Inventario` incrementandolo (`_it_Inventario ++`).

- El producto se encuentra en el inventario contrario:

  Si se encuentra, la secuencia siguiente es `"true"`:

```
if (ciudad_con_la_que_comerciaremos._it_Inventario
    != ciudad_con_la_que_comerciaremos._Inventario.
    end())
```

  Una vez dentro de la condición calculamos las diferencias de cada inventario, es decir, miramos las cantidades del producto que le sobran o necesita nuestra ciudad (`"this"`) y la ciudad con la que comerciamos. Si una de las dos ciudades necesita producto y la otra le puede proporcionar, se procederán a los cálculos que permitirán el intercambio de mercancía. En este intercambio calcularemos las cantidades, los nuevos pesos y volúmenes de los inventarios.

  Finalmente procederemos a mirar al siguiente producto del inventario de nuestra ciudad (`_it_Inventario ++`).

1.4 Paso 3 (invariante + condición de terminación):

La `invariante` del bucle es que al comienzo de cada iteración todos los productos procesados hasta ese momento están en un estado consistente y correcto en ambos inventarios, según las reglas de comercio.

La condición de terminación del bucle es:

```
_it_Inventario != this->_Inventario.end()}
```

Lo que significa que el bucle continuará mientras nuestro iterador no llegue al final del inventario de nuestra ciudad (`this->_Inventario`).

Dado que en cada iteración del bucle el iterador se incrementa con `_it_Inventario++`, cuando hayamos recorrido todo el inventario de

this, llegaremos al final del map, es decir, el iterador apuntará al elemento end de nuestro inventario. En ese momento, el bucle habrá recorrido todos los elementos de nuestro inventario, `L = 0...n`, donde `n` es el número de elementos.

Como hemos recorrido todos los elementos de L, el inventario de la ciudad con la que comerciamos, `C = 0...k`, con `k` elementos, también será modificado y recorrido parcialmente, solo los elementos comunes. Esto se debe a que en todo momento estamos buscando los elementos comunes y operando con ellos para realizar el intercambio de productos.

En conclusión, cuando el iterador apunta al elemento end, nuestro bucle habrá recorrido `n` elementos y habrá operado con `c` elementos, donde `c` es la cantidad de productos comunes y comerciables entre los inventarios `L` y `C`.

1.5 Paso 4 (número finito de iteraciones):

La expresión:

*_Inventario.size() - j [posición del elemento apuntado del **map**] + 1*

La expresión **no es negativa**, ja que `j` como máximo puede valer `_Inventario.size() - 1` y en cada iteración la expresión decrementa. Por lo tanto, la función **termina siempre y cuando el inventario no tenga infinitos elementos**. En conclusión, la **función está acotada** y, por lo tanto, es correcta.

1.6 Implementación de comerciar:

```
1   void Ciudad::comerciar_c(Cjt_Productos& Productos,
        Ciudad& ciudad_con_la_que_comerciaremos){
2     // Creamos un "iterator" que apunta al primer elemento
          del inventario
3     this->_it_Inventario = this->_Inventario.begin();
4     // Recordemos el inventario
5     while (this->_it_Inventario != this->_Inventario.end()
          )
6     {
7       // Miramos si el producto se encuentra dentro del
            inventario contrario
8       ciudad_con_la_que_comerciaremos._it_Inventario =
            ciudad_con_la_que_comerciaremos._Inventario.find(
            this->_it_Inventario->first);
9       // En caso afirmativo
10      if (ciudad_con_la_que_comerciaremos._it_Inventario
            != ciudad_con_la_que_comerciaremos._Inventario.
            end()){
```

```
11            // Calculamos la diferencia de productos de uno y
                  el otro
12            int diferencia_ciudad_1 = this->_it_Inventario->
                  second.second - this->_it_Inventario->second.
                  first;
13            int diferencia_ciudad_2 =
                  ciudad_con_la_que_comerciaremos._it_Inventario
                  ->second.second -
                  ciudad_con_la_que_comerciaremos._it_Inventario
                  ->second.first;
14            // En caso de que una ciudad necesite productos
                  que a la otra le sobran
15            if (diferencia_ciudad_1 * diferencia_ciudad_2 < 0)
                  {
16              // Si la ciudad 2 necesita productos que nuestra
                     ciudad le sobran
17              if (diferencia_ciudad_1 < 0){ // #Venta
18                if (diferencia_ciudad_2 > 0){
19                  // En caso de que tenemos m s productos de
                       los que necesita la ciudad 2
20                  if (abs(diferencia_ciudad_1) >= abs(
                       diferencia_ciudad_2)){
21                    // Consultamos las unidades pose das y
                         necesarias del producto que queremos
                         comerciar
22                    pair<int,int> producto = Productos.
                         consultar_producto_del_conjunto(this->
                         _it_Inventario->first);
23                    // Decrementamos en nuestra ciudad las
                         unidades que necesita la ciudad 2
24                    this->_it_Inventario->second.first -= abs(
                         diferencia_ciudad_2);
25                    this->_peso_total -= abs(
                         diferencia_ciudad_2) * producto.first;
26                    this->_volumen_total -= abs(
                         diferencia_ciudad_2) * producto.second;
27                    // Incrementamos en la ciudad 2 las
                         unidades que otorgamos la nuestra
                         ciudad
28                    ciudad_con_la_que_comerciaremos.
                         _it_Inventario->second.first += abs(
                         diferencia_ciudad_2);
29                    ciudad_con_la_que_comerciaremos.
                         _peso_total += abs(diferencia_ciudad_2)
                          * producto.first;
30                    ciudad_con_la_que_comerciaremos.
                         _volumen_total += abs(
                         diferencia_ciudad_2) * producto.second;
31                  }
```

```
32                  else { // En caso de que tenemos menos
                        productos de los que necesita la ciudad 2
33                    // Consultamos las unidades pose das y
                          necesarias del producto que queremos
                          comerciar
34                    pair<int,int> producto = Productos.
                          consultar_producto_del_conjunto(this->
                          _it_Inventario->first);
35                    // Decrementamos en nuestra ciudad todas
                          las unidades sobrantes
36                    this->_it_Inventario->second.first -= abs(
                          diferencia_ciudad_1);
37                    this->_peso_total -= abs(
                          diferencia_ciudad_1) * producto.first;
38                    this->_volumen_total -= abs(
                          diferencia_ciudad_1) * producto.second;
39                    // Incrementamos en la ciudad 2 las
                          unidades que sobran de nuestra ciudad
40                    ciudad_con_la_que_comerciaremos.
                          _it_Inventario->second.first += abs(
                          diferencia_ciudad_1);
41                    ciudad_con_la_que_comerciaremos.
                          _peso_total += abs(diferencia_ciudad_1)
                           * producto.first;
42                    ciudad_con_la_que_comerciaremos.
                          _volumen_total += abs(
                          diferencia_ciudad_1) * producto.second;
43                  }
44                }
45              }
46              // Si a nuestra ciudad necesita productos que la
                    ciudad 2 le sobran
47              else if (diferencia_ciudad_2 < 0){ // #Compra
48                if (diferencia_ciudad_1 > 0){
49                  // En caso de que la ciudad 2 tiene  m s
                        productos de los que necesita nuestra
                        ciudad
50                  if (abs(diferencia_ciudad_1) <= abs(
                        diferencia_ciudad_2)){
51                    // Consultamos las unidades pose das y
                          necesarias del producto que queremos
                          comprar
52                    pair<int,int> producto = Productos.
                          consultar_producto_del_conjunto(this->
                          _it_Inventario->first);
53                    // Incrementamos a nuestra ciudad todos
                          los productos que nos faltaban
54                    this->_it_Inventario->second.first += abs(
                          diferencia_ciudad_1);
```

```
55                  this->_peso_total += abs(
                        diferencia_ciudad_1) * producto.first;
56                  this->_volumen_total += abs(
                        diferencia_ciudad_1) * producto.second;
57                  // Decrementamos en la ciudad 2 las
                        unidades necesarias por nuestra ciudad
58                  ciudad_con_la_que_comerciaremos.
                        _it_Inventario->second.first -= abs(
                        diferencia_ciudad_1);
59                  ciudad_con_la_que_comerciaremos.
                        _peso_total -= abs(diferencia_ciudad_1)
                         * producto.first;
60                  ciudad_con_la_que_comerciaremos.
                        _volumen_total -= abs(
                        diferencia_ciudad_1) * producto.second;
61                }
62                else { // En caso de que la ciudad 2 tiene
                      menos productos de los que necesita
                      nuestra ciudad
63                  // Consultamos las unidades pose das y
                        necesarias del producto que queremos
                        comprar
64                  pair<int,int> producto = Productos.
                        consultar_producto_del_conjunto(this->
                        _it_Inventario->first);
65                  // Incrementamos a nuestra ciudad los
                        productos que le sobra a la ciudad 2
66                  this->_it_Inventario->second.first += abs(
                        diferencia_ciudad_2);
67                  this->_peso_total += abs(
                        diferencia_ciudad_2) * producto.first;
68                  this->_volumen_total += abs(
                        diferencia_ciudad_2) * producto.second;
69                  // Decrementamos en la ciudad 2 las
                        unidades sobrantes por esta ciudad
70                  ciudad_con_la_que_comerciaremos.
                        _it_Inventario->second.first -= abs(
                        diferencia_ciudad_2);
71                  ciudad_con_la_que_comerciaremos.
                        _peso_total -= abs(diferencia_ciudad_2)
                         * producto.first;
72                  ciudad_con_la_que_comerciaremos.
                        _volumen_total -= abs(
                        diferencia_ciudad_2) * producto.second;
73                }
74              }
75            }
76          }
77        }
78        // Consultamos el siguiente producto
```

```
79          _it_Inventario++;
80        }
81      }
```

Listing 2: comerciar_c (Ciudad)

2 Justificación de recorrido_con_max_beneficio

2.1 Especificación de recorrido_con_max_beneficio:

```
1   /**
2   * @brief Obtiene la ruta m s beneficiosa para el barco.
3   *
4   * @param nodo  rbol binario que representa la ciudad
        donde estamos localizados.
5   *
6   * @return Lista de nombres de ciudades.
7   *
8   * @pre Se proporciona el  rbol binario de la cuenca.
9   * @post Se obtiene la ruta m s beneficiosa para el
        barco.
10  */
11  list<pair<char,int>> recorrido_con_max_beneficio(const
        BinTree<pair<int,int>>& nodo);
```

Listing 3: recorrido_con_max_beneficio (Cuenca)

NOTA:

Nuestra función `recorrido_con_max_beneficio` recibe un árbol de pares fabricado anteriormente donde miramos si las ciudades del `río` (nodos) tiene o no los productos que el barco compra y vende.

Entonces, si el `pair<int,int>`; es igual a 0 quiere decir que el producto no se encuentra en la ciudad o que no puede comerciar con el barco. En caso contrario, la primera componente es la cantidad de producto que podemos comprar y la segunda la cantidad que podemos vender.

A partir de ahora `'recorrido_con_max_beneficio'` la llamaremos `func_rec`.

2.2 Paso 1 (Caso Base):

Si el *nodo es vacío*, retornamos una lista vacía porque no hay recorrido posible.

2.3 Paso 2 (H.I.):

Suponemos que func_rec produce resultados correctos para subárboles más pequeños que el árbol actual por la *hipótesis de inducción*.

Entonces, después de las llamadas recursivas **func_rec**(`nodo.left()`) y **func_rec**(`nodo.right()`), los recorridos '`recorrido_izquierda`' y '`recorrido_derecha`' contienen las listas de pares [**dirección, beneficio**] con el máximo beneficio acumulado desde los respectivos subárboles izquierdo y derecho.

2.4 Paso 3 (Paso Inductivo):

- Si el **subárbol izquierdo** está vacío, el beneficio acumulado de `recorrido_izquierda` es 0.
- Si el **subárbol derecho** está vacío, el beneficio acumulado de `recorrido_derecha` es 0.

Se calcula el beneficio del nodo actual como la suma del valor del nodo y el beneficio acumulado del camino seleccionado (izquierdo o derecho) con mayor beneficio.

La función luego compara los beneficios acumulados de ambos subárboles:

- Si el beneficio del **subárbol derecho es mayor**, se selecciona el camino derecho, se incrementa el beneficio del nodo actual, se añade el nodo actual al comienzo de la lista de `recorrido_derecha` y se retorna esta lista.
- Si el beneficio del **subárbol izquierdo es mayor**, se selecciona el camino izquierdo, se incrementa el beneficio del nodo actual, se añade el nodo actual al comienzo de la lista de `recorrido_izquierda` y se retorna esta lista.
- Si los **beneficios son iguales**, se selecciona el camino más corto; en caso de igualdad de longitud, se elige el camino izquierdo, se incrementa el beneficio del nodo actual, se añade el nodo actual al comienzo de la lista de `recorrido_izquierda` y se retorna esta lista.

Como las llamadas recursivas operan sobre subárboles más pequeños y se combinan correctamente los resultados, por la hipótesis de inducción y el correcto manejo de los beneficios, la función **func_rec** retorna una lista que representa el camino con el máximo beneficio acumulado.

2.5 Paso 4 (Terminación):

Función de Cota para func_rec:

Una función de cota para `func_rec` viene definida por la profundidad del nodo actual en el árbol y la distancia para llegar al final del árbol, que `disminuye` con cada llamada recursiva:

- **Profundidad del nodo**:
  - Si el nodo no es vacío, la **distancia disminuye en 1** con cada llamada recursiva a `nodo.left()` y `nodo.right()`. De tal manera que llegara un momento donde la **distancia sea 0** (llegamos al final del árbol) y el programa se encuentre con un **nodo vacío**.
  - Si el **nodo es vacío**, la **recursión termina** inmediatamente con el caso base retornando una lista vacía.

Por lo tanto, la recursión eventualmente llega al caso base siempre y cundo tengamos un número de ciudades finito, asegurando que la función `func_rec` **siempre termina**.

2.6 Implementación de func_rec:

```
1   list<pair<char,int>> Cuenca::recorrido_con_max_beneficio
        (const BinTree<pair<int,int>>& nodo){
2     // Caso Base ############################
3
4     // Si el nodo es vacio
5     if (nodo.empty()) return list<pair<char,int>>();
6
7     // Paso Inductivo #######################
8
9     // Construimos las listas postorden
10    list<pair<char,int>> recorido_izquierda =
          recorrido_con_max_beneficio(nodo.left());
11    list<pair<char,int>> recorido_derecha =
          recorrido_con_max_beneficio(nodo.right());
12    //---------------------------------
13
14    // Inicializamos los beneficios por ruta
15    int beneficio_izquierda = 0, beneficio_derecha = 0;
16
17    // Beneficios por ruta
18    if (!recorido_izquierda.empty()) beneficio_izquierda =
           recorido_izquierda.front().second;
19    if (!recorido_derecha.empty()) beneficio_derecha =
          recorido_derecha.front().second;
20    //--------------------
21
```

```
22        // Calculamos el beneficio en nuestra posici n
23        int beneficio_nodo_actual = nodo.value().first + nodo.
              value().second;
24
25        // Distinguimos por diferentes casos:
26
27        // En caso de que el beneficio de la parte izquierda y
               la parte derecha sea 0, entonces finalizamos la
              funci n
28        if (beneficio_izquierda + beneficio_derecha == 0) {
29          // Devolvemos la lista formada  nicamente por un
                elemento
30          list<pair<char,int>> lista_unica;
31          // Char u -> Representa una  nica ciudad en la
                lista
32          lista_unica.push_front(make_pair('u',
                beneficio_nodo_actual));
33          return lista_unica;
34        }
35
36        // Si el beneficio de la derecha es mayor que el de la
               izquierda
37        if (beneficio_derecha > beneficio_izquierda){
38          // Si beneficio_derecha   s  mayor que
                beneficio_izquierda quiere decir que es mayor que
                 0 en el peor de los casos cosa que quiere decir
                que existe.
39
40          // Incrementamos el beneficio
41          beneficio_nodo_actual += recorido_derecha.front().
                second;
42          // A adimos el nuevo nodo y la direcci n por la
                que debemos ir
43          recorido_derecha.push_front(make_pair('d',
                beneficio_nodo_actual));
44          // Devolvemos el recorrido hecho
45          return recorido_derecha;
46
47        } // Si el beneficio de la izquierda es mayor que el
              de la derecha
48        else if (beneficio_derecha < beneficio_izquierda){
49          // Si beneficio_izquierda   s  mayor que
                beneficio_izquierda quiere decir que es mayor que
                 0 en el peor de los casos cosa que quiere decir
                que existe.
50
51          // Incrementamos el beneficio
52          beneficio_nodo_actual += recorido_izquierda.front().
                second;
```

```
53          // A adimos el nuevo nodo y la direcci n por la
                que debemos ir
54          recorido_izquierda.push_front(make_pair('i',
                beneficio_nodo_actual));
55          // Devolvemos el recorrido hecho
56          return recorido_izquierda;
57        }
58        else { // Si el beneficio de la izquierda es igual que
              el de la derecha
59          // Miramos el recorrido  m s  corto de los dos, si
                son iguales escogeremos el de la izquierda
60          // Si el recorrido de la derecha es  m s  peque o
                que el de la izquierda
61          if (recorido_derecha.size() < recorido_izquierda.
                size()){
62            // Entonces, como existe el recorrido por la
                  derecha: (Explicado arriba)
63            // Incrementamos el beneficio
64            beneficio_nodo_actual += recorido_derecha.front().
                  second; //Incrementamos el beneficio
65            // A adimos el nuevo nodo y la direcci n por la
                  que debemos ir
66            recorido_derecha.push_front(make_pair('d',
                  beneficio_nodo_actual));
67            // Devolvemos el recorrido hecho
68            return recorido_derecha;
69          }
70          else { // Si el recorrido de la izquierda es  m s
                peque o  o igual que el de la derecha
71            // Entonces, si existe el recorrido por la
                  izquierda
72            // Incrementamos el beneficio
73            if (!recorido_izquierda.empty())
                  beneficio_nodo_actual += recorido_izquierda.
                  front().second; //Incrementamos el beneficio
74            // A adimos el nuevo nodo y la direcci n por la
                  que debemos ir
75            recorido_izquierda.push_front(make_pair('i',
                  beneficio_nodo_actual));
76            // Devolvemos el recorrido hecho
77            return recorido_izquierda;
78          }
79        }
80      }
```

Listing 4: recorrido_con_max_beneficio (Cuenca)