Universidad Católica del Uruguay

Algoritmos y Estructuras de Datos I

CASO DE ESTUDIO – PARTE II

VERONICA ECHEZARRETA

Contenido

Introducción

Se realizó una simulación de una cadena de supermercados y su funcionamiento, ya sea agregando productos a la empresa, y a su vez, estos mismos productos a cada sucursal, así como eliminando productos de cada sucursal o listando los productos existentes en cada sucursal junto con su stock, etc.

Problema planteado

Al simular el funcionamiento de una cadena de supermercados, se requieren distintas funcionalidades:

- Incorporar un nuevo producto a una sucursal del supermercado.
- Agregar stock a un producto existente en una sucursal.
- Simular la venta de un producto en una sucursal (reducir el stock de un producto existente). De no haber stock suficiente para la venta en esa sucursal, deberá indicarse la lista de sucursales que tengan el stock suficiente, ordenada por cantidad de producto.
- Eliminar productos que ya no se venden (por no ser comercializados más) en todas las sucursales del supermercado.
- Dado un código de producto, indicar las existencias de este en todas las sucursales, ordenada por sucursal.
- Listar todos los productos registrados, en una sucursal, ordenado por nombre de producto, presentando además su stock.
- Listar todos los productos registrados, ordenados por ciudad, barrio, y nombre de producto, presentando además su stock.

Cambios respecto a la parte I

Propuesta original

La propuesta original supone el uso de arboles binarios de búsqueda para almacenar los productos en la empresa.

También se utilizan estos mismos para almacenar las sucursales en la empresa, y estas sucursales, a su vez, emplean árboles para almacenar los productos en cada una de ellas.

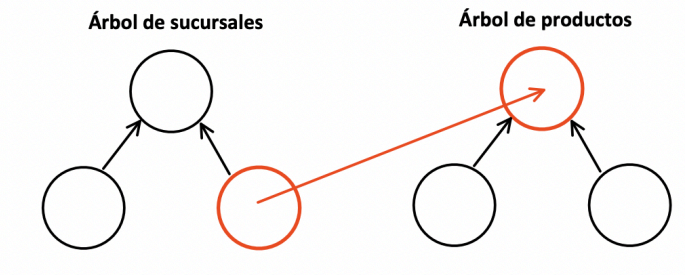
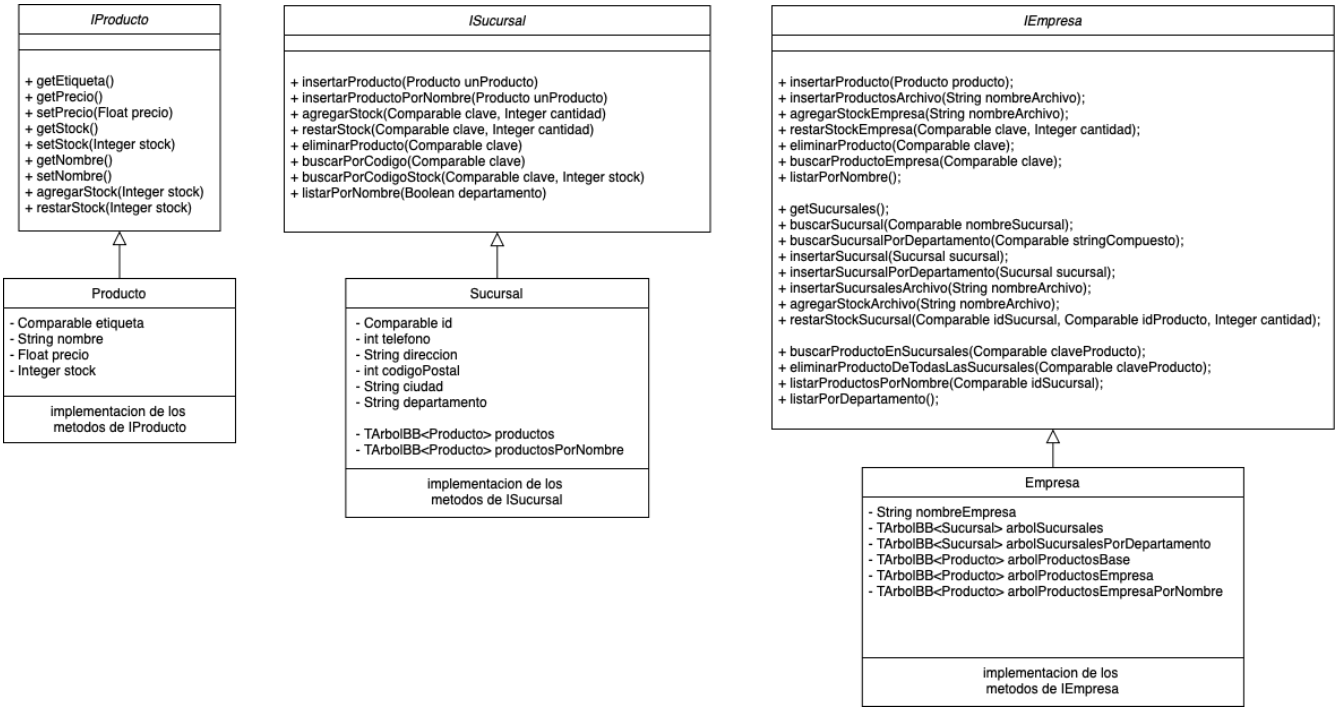


Diagrama de clases



Nueva propuesta

La nueva propuesta supone el uso de arboles AVL para almacenar los productos y las sucursales en la empresa, así como también los productos de cada sucursal.

Al utilizar arboles AVL, la busqueda mantiene un orden de complejidad O(log n), por lo que es deseado sobre todo en casos donde se van a insertar muchos elementos, como lo es en el caso de insertar productos en cada sucursal y en la empresa.

En la sección siguiente se verá a fondo el desarrollo de estos nuevos algoritmos comparados con los algoritmos originales. En el programa se han cambiado las implementaciones de “insertar” a “insertarBalanceado” igualmente que de “eliminar” a “eliminarBalanceado”.

Algoritmos

Métodos auxiliares necesarios para insertar y eliminar balanceado

Nuevo atributo de TElementoAB:

- int altura

Crear una nueva altura para los elementos

Lenguaje natural:
Se inicializan dos enteros en -1, uno para la altura del subárbol izquierdo y otro para el subárbol derecho. Si alguno de los dos hijos no es nulo, se toma la altura de estos y al final, se setea la altura de la raíz tomando la máxima entre los dos subárboles y sumándole 1.

Precondiciones:
- El elemento no puede ser nulo

Postcondiciones:
- La altura deberá ser el máximo de los dos

Pseudocódigo:

TElementoAB.nuevaAltura() : void	**O(1)**
COM	
entero izq ← -1	O(1)
entero der ← -1	O(1)
SI (hijoIzq <> nulo) ENTONCES	O(1)
izq ← hijoIzq.altura	O(1)
FIN SI	
SI (hijoDer <> nulo) ENTONCES	O(1)
der ← hijoDer.altura	O(1)
FIN SI	
altura ← 1 + max(izq, der)	O(1)
FIN	

Crear una diferencia de altura para los elementos

Lenguaje natural:
Se crean dos enteros, uno para la altura del subárbol derecho y otro para la altura del subárbol izquierdo. Si los hijos no son nulos, se toma la altura de estos y se pasa al entero correspondiente. Si no tuviera alguno de los hijos, la altura de ese subárbol pasaría a ser -1. Por ultimo, se devuelve la resta entre la altura del subárbol derecho y el subárbol izquierdo.

Precondiciones:

- El elemento no puede ser nulo

Postcondiciones:

- La diferencia entre alturas no debería ser mayor a 2

Pseudocódigo:

TElementoAB.diferenciaAltura(TElementoAB elem) : entero **O(1)**
COM

```
    entero alturaDerecha                                    O(1)
    entero alturaIzquierda                                  O(1)

    SI (elem.hijoDer <> nulo) ENTONCES                      O(1)
        alturaDerecha ← elem.hijoDer.altura                 O(1)
    SINO
        alturaDerecha ← -1                                  O(1)
    FIN SI

    SI (elem.hijoIzq <> nulo) ENTONCES                      O(1)
        alturaIzquierda ← elem.hijoIzq.altura               O(1)
    SINO
        alturaIzquierda ← -1                                O(1)
    FIN SI

    devolver (alturaDerecha – alturaIzquierda)              O(1)
```

FIN

Balancear el árbol

Lenguaje natural:

Se inicializa un entero llamado “diferencia” tomando en cuenta el método creado anteriormente. Si la diferencia llegara a ser de -2, se pregunta si la altura del hijo izquierdo de este hijo izquierdo es mayor a la altura del hijo derecho de este hijo izquierdo, y si la respuesta fuera positiva, se realiza una rotación LL (ya visto anteriormente en clase) del elemento, en caso contrario, se realiza una rotación LR. Si la diferencia fuera de 2, se pregunta si la altura del hijo derecho de su hijo derecho es mayor a la altura del hijo izquierdo del mismo hijo derecho. Si fuera afirmativo, se realiza una rotación RR, en caso contrario, una rotación RL.

Precondiciones:

- El elemento no puede ser nulo

Postcondiciones:

- Se devuelve el elemento correctamente posicionado en el árbol

Pseudocódigo:

TElementoAB.balancearArbol() : TElementoAB **O(1)**
COM

```
    entero diferencia ← diferenciaAltura(this)                    O(1)

    SI (diferencia = -2) ENTONCES                                 O(1)
        SI (hijoIzq.hijoIzq.altura > hijoIzq.hijoDer.altura) ENTONCES   O(1)
            devolver rotacionLL(this)                             O(1)
        SINO
            devolver rotacionLR(this)                             O(1)
        FIN SI
    SINO (diferencia = 2) ENTONCES                                O(1)
        SI (hijoDer.hijoDer.altura > hijoDer.hijoIzq.altura) ENTONCES   O(1)
            devolver rotacionRR(this)                             O(1)
        SINO
            devolver rotacionRL(this)                             O(1)
        FIN SI
    FIN SI

    devolver this                                                 O(1)

FIN
```

Métodos para insertar y eliminar balanceado

Insertar un elemento balanceado

Lenguaje natural:
Se inserta un elemento al árbol de manera que este mismo quede balanceado, es decir, que cada uno de sus subárboles tenga una diferencia de altura máxima de 1.

Precondiciones:
- Debe ser un elemento valido

Postcondiciones:
- El árbol de productos tendrá un elemento más (o no si es un elemento que ya existía en este)

Pseudocódigo:

```
TArbolBB.insertarBalanceado(TElementoAB elem) : void      O(log n)
COM
    SI (esVacio) ENTONCES                                 O(1)
        raíz ← elem                                       O(1)
    SINO
        raiz.insertarBalanceado(elem)                     O(log n)
    FIN SI
FIN


TElementoAB.insertarBalanceado(TElementoAB elem) : TElementoAB   O(log n)
COM
    SI (elem.etiqueta < this.etiqueta) ENTONCES           O(1)
        SI (hijoIzq <> nulo) ENTONCES                     O(1)
```

```
            hijoIzq ← hijoIzq.insertarBalanceado(elem)          O(1)
            devolver balancearArbol()                           O(log n)
        SINO
            hijoIzq ← elem                                      O(1)
            devolver this                                       O(1)
        FIN SI
    SINO SI (elem.etiqueta > this.etiqueta) ENTONCES            O(1)
        SI (hijoDer <> nulo) ENTONCES                           O(1)
            hijoDer ← hijoDer.insertarBalanceado(elem)          O(1)
            devolver balancearArbol()                           O(log n)
        SINO
            hijoDer ← elem                                      O(1)
            devolver this                                       O(1)
        FIN SI
    SINO
        devolver this                                           O(1)
    FIN SI
FIN
```

Eliminar un elemento balanceado

Lenguaje natural:
Se busca un elemento en el árbol mediante una clave dada por parámetro. Si ese elemento existe en ese árbol, se elimina, sino se devuelve falso.

Precondiciones:

- La clave por la que se busca debe ser una clave válida

Postcondiciones:

- El árbol tendrá un elemento menos o seguirá con la misma cantidad de elementos.

Pseudocódigo:

```
TArbolBB.eliminarBalanceado(Comparable unaEtiqueta) : void          O(log n)
COM
    SI (NOT esVacio) ENTONCES                                       O(1)
        this.raíz ← this.raíz.eliminarBalanceado(unaEtiqueta)       O(log n)
    FIN SI
FIN
```

```
TElementoAB.eliminarBalanceado (Comparable unaEtiqueta) : TElementoAB   O(log n)
COM
    SI (unaEtiqueta < this.etiqueta) ENTONCES                       O(1)
        SI (hijoIzq <> nulo) ENTONCES                               O(1)
            hijoIzq ← hijoIzq.eliminarBalanceado(unaEtiqueta)       O(1)
            devolver balancearArbol()                               O(log n)
        FIN SI
        devolver this                                               O(1)
```

```
    SINO SI (unaEtiqueta > this.etiqueta) ENTONCES                  O(1)
        SI (hijoDer <> nulo) ENTONCES                               O(1)
            hijoDer ← hijoDer.eliminarBalanceado(unaEtiqueta)       O(1)
            devolver balancearArbol()                               O(log n)
        FIN SI
        devolver this                                               O(1)
    FIN SI

    TElementoAB nuevo ← quitaElNodo()                               O(1)
    devolver nuevo                                                  O(1)
FIN
```

Selección y justificación de cambios a implementar

Se eligió el uso de arboles AVL en comparación con arboles binarios de búsqueda, ya que estos últimos tenían un orden de complejidad O (N) en el peor de los casos cuando se insertaban y eliminaban elementos, mientras que los arboles AVL tienen un orden de complejidad O (log n) en el peor de los casos para los mismos métodos.

Conclusiones

- Es difícil agrupar la información en una única estructura y esperar que el acceso a la información sea óptimo para todos los casos de uso (ejemplo: sucursales ordenadas por nombre vs. sucursales ordenadas por departamento, ciudad y código postal).
- Puede aprovecharse la existencia de punteros a objetos para poder mantener diferentes estructuras (en este caso varios ABB con diferentes criterios de ordenación) en memoria sin la necesidad de duplicar los objetos en sí, evitando malgastar memoria.
- El uso de claves compuestas para los árboles ayuda a que la información pueda ser ordenada en un árbol por varios criterios anidados, donde el primer atributo se repite entre varios objetos.

Guía del usuario

El programa consta de dos partes: en la primera se ejecutan los métodos mismos de la empresa, en la segunda, se ejecutan los métodos de la empresa con las sucursales.

Para que el programa corra más rápido, si se quiere, se comentan los métodos que no se utilicen en el momento y se descomentan los que si se quieran utilizar. Por ejemplo, si se quieren ejecutar los métodos de la primera parte, se comentan los métodos de la segunda parte, y viceversa.